Conferences in Research and Practice in Information Technology

Volume 160

Computing Education 2015

Australian Computer Science Communications, Volume 37, Number 2

Computing Education 2015

Proceedings of the 17th Australasian Computing Education Conference (ACE 2015),
Sydney, Australia, 27 - 30 January 2015

Daryl D'Souza and Katrina Falkner, Eds.

Volume 160 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 2015. Proceedings of the 17th Australasian Computing Education Conference (ACE 2015), Sydney, Australia, 27 - 30 January 2015

Conferences in Research and Practice in Information Technology, Volume 160.

Editors:

Dr Daryl D'Souza
School of Computer Science and Information Technology
RMIT University
GPO Box 2476
Melbourne VIC 3001
Australia
Email: `daryl.dsouza@rmit.edu.au`

Associate Professor Katrina Falkner
School of Computer Science
University of Adelaide
Adelaide SA 5005
Australia
Email: `katrina.falkner@adelaide.edu.au`

Series Editors:
Vladimir Estivill-Castro, Griffith University, Queensland
Simeon J. Simoff, University of Western Sydney, NSW
Email: `crpit@scem.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 160.
ISSN 1445-1336.
ISBN 978-1-921770-42-5.

Document engineering, January 2015 by CRPIT
On-line proceedings, January 2015 by the University of Western Sydney
Electronic media production, January 2015 by Laurence Park, University of Western Sydney

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

Preface

Welcome to the Seventeenth Australasian Computing Education Conference (ACE2015). This year the ACE2015 conference, which is part of the Australasian Computer Science Week, is being held at University of Western Sydney (Parramatta Campus), Sydney, Australia, from 27 to 30 January, 2015.

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 20 full papers from the 42 papers submitted, giving an acceptance rate of 47%. The number of submissions was slightly more than the 39 submitted in the previous year, but once again with a strong national and international presence. Accepted papers reflected a variety of topics, with the paper sessions organized into themes, accordingly as: (1) Gender, Curriculum, Employment; (2) ICT Education I; (3) ICT Education II; (4) Programming Assessment; (5) Introductory/Novice Programming. As usual many of the papers present new innovations and demonstrate high quality research.

The doctoral consortium is chaired by Dr Claudia Szabo from the University of Adelaide, Australia. As with past ACE conferences, we are continuing to hold workshops. This year two workshops have been organised, both of these led by Associate Professor David Klappholz, Stevens Institute of Technology (New Jersey, USA). Details are as follows:

– Tutorial WorkshopReal Projects for Real Clients Capstone Course
– Research WorkshopDeveloping a Concept Inventory for Discrete Mathematics

Best papers are awarded on the basis of the double blind peer reviews of the paper and were selected by the senior co-chair Dr. Jacqueline Whalley. This year ACE awarded a best paper and best student paper. The best paper was awarded to:

– What are we doing when we assess programming?
  Dale Parsons, Krissi Wood and Patricia Haden

One other paper was also highly commended:

– Teaching Computational Thinking in K-6: The CSER Digital Technologies MOOC
  Katrina Falkner, Rebecca Vivian and Nickolas Falkner.

The best student paper was awarded to:

– Comparative Study on Programmable Robots as Programming Educational Tools
  Shohei Yamazaki, Kazunori Sakamoto, Kiyoshi Honda, Hironori Washizaki and Yoshiaki Fukazawa

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, SGI, our hosts University of Western Sydney, Australia, and the Australasian Computing Education executive for the opportunity to chair the ACE2015 conference.

Daryl D'Souza
RMIT University

Katrina Falkner
University of Adelaide

ACE 2015 Conference Co-chairs
January 2015

Programme Committee

Chairs

Daryl D'Souza, RMIT University, Australia
Katrina Falkner, The University of Adelaide, Australia

Members

Bradley Alexander, The University of Adelaide, Australia
Matthew Butler, Monash University, Australia
Mats Daniels, Uppsala University, Sweden
Michael De Raadt, Moodle, Australia
Paul Denny, The University of Auckland, New Zealand
Julian Dermoudy, University of Tasmania, Australia
John Hamer, The University of Auckland, New Zealand
Margaret Hamilton, RMIT University, Australia
Chris Johnson, ANU, Australia
Mikko Laakso, University of Turku, Finland
Andrew Luxton-Reilly, University of Auckland, NZ
Raina Mason, Southern Cross University, Australia
Chris McDonald, University of Western Australia, Australia
Michael Morgan, Monash University, Australia
Dale Parsons, Otago Polytechnic, New Zealand
Arnold Pears, Uppsala University, Sweden
Anne Philpott, AUT University, New Zealand
Helen Purchase, The University of Glasgow, UK
Anthony Robins, University of Otago, New Zealand
Judy Sheard, Monash University, Australia
Simon, University of Newcastle, Australia
Claudia Szabo, The University of Adelaide, Australia
Charles Thevathayan, RMIT University
Josh Tenenberg, University of Washington, USA
Errol Thompson, Aston University, United Kingdom

Conference Webmaster

Daryl D'Souza, RMIT University, Australia.

Organising Committee

Chairs

Professor Athula Ginige, University of Western Sydney, Australia
Associate Professor Paul Kennedy, University of Technology, Sydney, Australia

Local Chair

Dr Bahman Javadi, University of Western Sydney, Australia

Publicity Chair

Dr Ante Prodan, University of Western Sydney, Australia

Publication Chair

Dr Laurence Park, University of Western Sydney, Australia

Finance Chair

Michael Walsh, University of Western Sydney, Australia

Sponsorship Chair

Kerry Holling, University of Western Sydney, Australia

Doctoral Consortia Co-chairs

Professor Anthony Maeder, University of Western Sydney, Australia
Dr Siamak Tafavogh, University of Technology, Sydney, Australia

Event Coordinator

Nicolle Fowler, University of Western Sydney, Australia

Welcome from the Organising Committee

On behalf of the Organising Committee, it is our pleasure to welcome you to Sydney and to the 2015 Australasian Computer Science Week (ACSW 2015). This year the conference is hosted by the University of Western Sydney and it's School of Computin,g Engineering and Mathematics.

A major highlight of the ACSW 2015 will be the Industry Research Nexus day on 27th January 2015. The aim is for industry leaders and academic researchers to come together and explore research areas of mutual interest. Many University research groups and 15 industries have confirmed their participation.

ACSW 2015 consists of 9 sub conferences covering a range of topics in Computer Science and related areas. These conferences are:

- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Motoshi Saeki and Henning Köhler)
- Australasian Computer Science Conference (ACSC) (Chaired by Dave Parry)
- Australasian Computing Education Conference (ACE) (Chaired by Daryl D'Souza and Katrina Falkner)
- Australasian Information Security Conference (AISC) (Chaired by Ian Welch and Xun Yi)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Bahman Javadi and Saurabh Garg)
- Australasian User Interface Conference (AUIC) (Chaired by Stefan Marks and Rachel Blagojevic)
- Australasian Web Conference (AWC) (Chaired by Joseph Davis)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Anthony Maeder and Jim Warren)
- Interactive Entertainment (IE) (Chaired by Yusuf Pisan and Keith Nesbitt)

Social events are a very important part of a conference as these provide many networking opportunities. To foster networking we have included a reception with industry on 27th January 2015, a Welcome reception on 28th January 2015 and a conference dinner on 29th January 2015.

Organising a multi-conference event such as ACSW is a challenging process even with many hands helping to distribute the workload, and actively cooperating to bring the events to fruition. This year has been no exception. We would like to share with you our gratitude towards all members of the organising committee for their combined efforts and dedication to the success of ACSW2015. We also thank all conference co-chairs and reviewers, for putting together the conference programs which are the heart of ACSW, and to the organisers of the sub conferences, workshops, poster sessions and Doctoral Consortium. Special thanks to John Grundy as chair of CoRE for his support for the innovations we have introduced this year.

This year we have secured generous support from several sponsors to help defray the costs of the event and we thank them for their welcome contributions. Last, but not least, we would like to thank all speakers, participants and attendees, and we look forward to several days of stimulating presentations, debates, friendly interactions and thoughtful discussions.

Athula Ginige
University of Western Sydney

Paul Kennedy
University of Technology Sydney

ACSW2015 General Co-Chairs
January, 2015

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2015 in Sydney. 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 component conferences of ACSW have changed over time with additions and subtractions: ACSC, ACE, AISC, AUIC, AusPDC, HIKM, ACDC, APCCM, CATS and AWC. Two doctoral consortia (ACDC and ACE-DC) and an Australasian Early Career Researchers Workshop (AECRW) reflect the evolving dimensions of ACSW and build on the diversity of the Australasian computing community. A specific industry day on the 27th January to facilitate academic / industry discussion and networking is a key feature of ACSW 2015.

In 2015, we are fortunate to have Professor Omer Rana, Associate Professor Pascal Hitzler and Professor Mark Sagar providing keynote talks to the conference. I thank them for their contributions to ACSW2015.

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 Professor Athula Ginige, Professor Paul Kennedy and their colleagues for organising what promises to be a vibrant event. Below I outline some of CORE's activities in 2013/14.

I welcome feedback on these including other activities you think CORE should be active in.

The major sponsor of Australian Computer Science Week:
- The venue for the annual Heads and Professors meeting
- An opportunity for Australian & NZ computing staff and postgrads to network and help develop their research and teaching
- Substantial discounts for attendees from member departments
- A doctoral consortium at which postgrads can seek external expertise for their research
- An Early Career Research forum to provide ECRs input into their development

Sponsor of several research, teaching and service awards:
- Chris Wallace award for Distinguished Research Contribution
- CORE Teaching Award
- Australasian Distinguished Doctoral Dissertation
- John Hughes Distinguished Service Award
- Various "Best Student Paper" awards at ACSW

Development, maintenance, and publication of the CORE conference and journal rankings. In 2014 this includes a heavily-used web portal with a range of holistic venue information and a community update of the CORE 2013 conference rankings.

Input into a number of community resources and issues of interest:
- Development of an agreed national curriculum defining Computer Science, Software Engineering, and Information Technology
- A central point for discussion of community issues such as research standards
- Various submissions on behalf of Computer Science Departments and Academics to relevant government and industry bodies, including recently on Australian Workplace ICT Skills development, the Schools Technology Curriculum and the Defence Trade Controls Act.

Coordination with other sector groups:
- Work with the ACS on curriculum and accreditation
- Work with groups such as ACDICT, ACPHIS and government on issues such as CS staff performance metrics and appraisal, and recruitment of students into computing
- A member of CRA (Computing Research Association) and Informatics Europe. These organisations are the North American and European equivalents of CORE.
- A member of Science & Technology Australia, which provides eligibility for Science Meets Parliament and opportunity for input into government policy, and involvement with Science Meets Policymakers

The 2014 Executive Committee has been looking at a range of activities that CORE can lead or contribute to, including more developmental activities for CORE members. This has also included a revamp of the mailing lists, web site, creation of discussion forums, identification of key issues for commentary and lobbying, and working with other groups to attract high aptitude students into ICT courses and careers.

Again, I welcome your active input into the direction of CORE in order to give our community improved visibility and impact. 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 2014, and look forward to the continuing shaping and development of the Australasian computing community in 2015.

John Grundy

President, CORE
January, 2015

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.

2016. Volume 38. Host and Venue - Australian National University, Canberra, ACT.

2015. Volume 37. Host and Venue - University of Western Sydney, NSW.

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 Computing 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
IE	Australasian Conference on Interactive Entertainment

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

ACSW and ACE 2015 Sponsors

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

Host Sponsors

University of Western Sydney
www.uws.edu.au

Australian Computer Society
www.acs.org.au

Computing Research and Education
www.core.edu.au

Platinum Sponsor

Dimension Data
www.dimensiondata.com

Gold Sponsors

NTT Australia Pty Ltd
www.au.ntt.com

Hewlett-Packard Company
www.hp.com

Intersect
www.intersect.org.au

Cognizant Technology Solutions
www.cognizant.com

Silver Sponsors

SGI
www.sgi.com

SMS Management and Technology
www.smsmt.com

AARNet
www.aarnet.edu.au

Macquarie Telecom
www.macquarietelecom.com

Bronze Sponsors

Australian Access Federation
aaf.edu.au

NEC Australia Pty Ltd
au.nec.com

Squiz Australia
www.squiz.net/au

Talent RISE
www.talentrise.org

Espire Infolabs Pty Ltd
www.espire.com

Contributed Papers

Gender differences in experiences of TAFE IT students: a work in progress

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

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

Tim Comber
Southern Cross University
tim.comber@scu.edu.au

Anne Hellou
North Coast Institute of TAFE
Anne.Hellou@det.nsw.edu.au

Julie Tucker
Southern Cross University
julie.tucker@scu.edu.au

Abstract

In Australia, one of the sources of loss of females in the IT education pipeline occurs at the TAFE (college) level. Female students comprise the majority of early TAFE IT courses and female completion rates for these courses are similar to males. Despite this early success, most females choose to not continue to Diploma level, and through articulation pathways into university IT courses. A survey was conducted to determine possible differences in experiences between male and female TAFE IT students. It was found that more females than males lived alone or with dependents. Female students had higher employment status, higher previous education, and comparable computer literacy and interest in IT to the male cohort. Despite these advantages, the female students had lower confidence in their ability to study, and their abilities in IT, and many female students did not intend to study or work in IT. Possible reasons are discussed.

Keywords: Women in Technology, attrition, self-efficacy, survey, TAFE, college.

1 Introduction

Since 2008 the Women in Technology (WIT) program at Southern Cross University has been conducting events such as games nights, robotics workshops, and social events to attract female students to study IT, and to support and retain these students throughout their course. The WIT program's purpose is to address the low proportion of females in IT courses and the IT industry - currently around 15% in tertiary educational institutions and around 22% in the IT workforce (Australian Computer Society 2011). The narrowing of the educational pipeline of females studying IT is likely to lead to a greater gender imbalance in the future, and the lack of participation by females in the production of technology has an ongoing impact on the shaping and content of that technology (Logan & Crump 2007).

A recent WIT think-tank - involving university staff, students, TAFE NSW staff, and representatives of local IT industries and employment agencies - identified that one of the sources of loss of females in the IT education pipeline occurred at the TAFE (technical college) level. Specifically, TAFE NSW North Coast Institute (NCTAFE) offers various Certificate, Diploma and Advanced Diploma courses and other training packages in IT, some of which were articulation programs into undergraduate computing degrees at Southern Cross University. NCTAFE comprises campuses in Northern NSW, with approximately 1650 students studying IT courses (Certificate 1 through to Advanced Diploma) each year.

Within the TAFE IT courses, a substantial number of female students who start IT Certificates do not progress to Diploma or Advanced Diploma courses. For example, in 2012, across all campuses and study modes, 134 female students (vs 81 male students) enrolled in Certificate 1 IT courses but this number declined to just 1 female (and 22 male students) in the Advanced Diploma courses. Surprisingly, there are actually more females than males in Certificate 1 IT courses but for the Advanced Diplomas the proportion drops to less than 5% females (Figure 1).

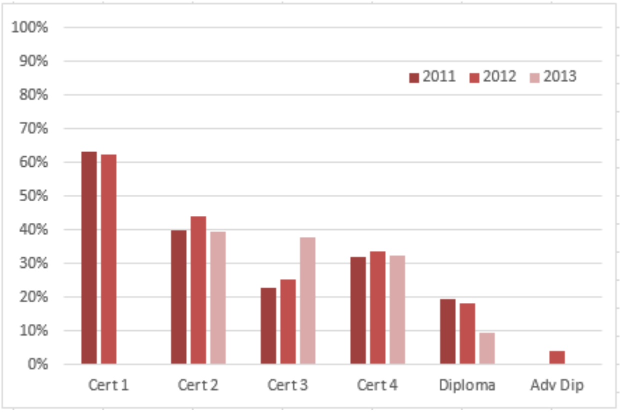


Figure 1: Percentage of female students in NCTAFE IT courses (Data provided by NCTAFE 2013)

Females are not leaving further study at TAFE in the same proportions in other courses. In fact, in certification levels across all courses in TAFE NSW, females comprise over 45%, and higher than 50% in Certificate 4 and Diploma (TAFE NSW 2013). This is illustrated in Figure 2 with the enrolment data for 2012.

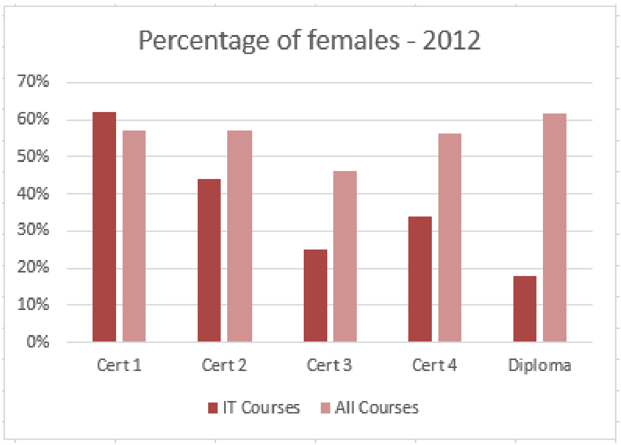


Figure 2: Percentages of female students in IT courses vs All courses (TAFE NSW)

This decrease in number of female students between Certificate 1 IT courses and Advanced Diplomas should not be seen as an indication that female students are less able than males at IT. Completion rates for each course are similar for males and females (Hellou 2013). At the end of each course, students are required to make a conscious decision to pursue further study, and to enrol in the next course. Fewer female students are making the choice to continue in IT than are male students.

If this decrease in female students between initial enrolment and final qualification was happening in a university undergraduate course, it would present as withdrawals or failures and would be subject to scrutiny to find out what was happening to these female students. The TAFE course configuration allows many exit points and so this 'choose not to progress' attrition has not been previously identified. The direct result is that the pipeline of female IT students entering universities through articulation pathways from TAFE is narrowed. In 2012 at Southern Cross University, for example, females comprised merely 9% of the TAFE IT articulation pathway intake compared to 13.5% in the normal educational intake (school-leavers and mature-age students). The reasons for this decrease in female enrolment percentages from TAFE IT Certificate courses to Diploma courses are not currently known. Before programs can be put in place that aim to boost the numbers of females continuing to study IT at TAFE or University, the reasons why female IT students choose not to continue need to be determined.

Many reasons have been proposed for the low numbers of female IT students, including external factors such as family obligations and obtaining employment, and other factors such as lower self-confidence, lower initial computer literacy, lack of availability of desired choice of Diploma programs and societal gender expectations (Cohoon 2001; Katz et al. 2006; Roberts et al. 2012; West & Ross 2002). A survey was proposed as a starting point to determine the differences, if any, between male and female IT students' experiences, life circumstances and attitudes.

2 Methodology

In early 2014, all NCTAFE Information Technology students were invited to participate in an online survey, with the objective of determining if there were differences in the profile of male and female students in IT courses that could impact on their progression. A wide range of questions were asked in the survey, to identify any possible differences between male and female experiences and attitudes.

Some of the areas examined were:

- age group;
- living circumstances (living alone, with family, share house etc.);
- primary caregiver status (of children or others);
- number of dependents;
- employment status;
- financial pressure experienced;
- current level of emotional well-being;
- prior educational level;
- use of computers;
- previous study in IT;
- reasons for the choice of course;
- attitudes towards study and the course, including self-efficacy; and
- future intentions in study and employment.

The full set of questions may be accessed at http://bit.ly/NCITSurveyQuestions.

NCTAFE offers courses in on-campus, distance/online and 'mixed' mode. Online students can commence a course at any time during the year. For this reason, the invitation to participate was sent to all new students with their enrolment package, as well as invitations and information disseminated to all current IT students.

3 Results

3.1 Responses

The survey was open for 2 weeks, and collected 78 responses in total, which is about 10% of the Semester 1 IT course enrolments. One of these respondents was under 18 years of age, and this response was discarded, as approval was not obtained from a parent or guardian. Several participants did not progress past the first two questions of the survey - age and gender - and these responses were not included in the data set.

There were a total of 65 complete or mostly-complete responses included in the analyses below. Where not all participants chose to answer a question, this has been indicated in the results.

3.2 Demographics

Gender: Thirty-seven males (57%) and twenty-eight females (43%) participated in the survey. It should be noted that while there were more male than female participants, the purpose of the survey was to examine the differences in circumstances and attitudes between genders, and for this purpose the proportion of male to female participants is suitable.

Age: The age groups of the survey participants are shown below in Table 1.

Approximately 66% of the participants were over 40 years of age. Although NCTAFE is a regional institution which does have a large amount of mature-age students, the proportion of older students in this sample is higher than expected compared to the general population of NCTAFE students (NSW Dept of Education and Communities 2012).

	Total	Male	Female
18-21	6	6	0
22-25	6	4	2
26-30	4	1	3
31-40	6	5	1
41-50	23	11	12
51-60	10	5	5
over 60	10	5	5

Table 1: Age of survey participants

The age distributions for the participants in this sample, compared to the general NCTAFE population, should be kept in mind when considering the results of this survey.

3.3 Current Course of Study

Students may enter a TAFE pathway of study at any level, if they have satisfied the entry requirements. For example, some students may be studying a Certificate 4 as their first TAFE IT course, or they may be in Certificate 2 as their second IT course at TAFE. The numbers in our sample in each course are shown below in Table 2. Participation rates are dominated by Certificate 4 students, followed by Diploma students, and these two combined account for a total of 45 out of the 65 participants (69%) that answered this question.

Current Course	Total	Male	Female
Cert 1	1	0	1
Cert 2	6	4	2
Cert 3	11	7	4
Cert 4	28	15	13
Diploma	17	10	7
Adv. Dip.	0	0	0

Table 2: Current TAFE IT course level

Approximately 45% of our sample (50% of males and 41% of females) were enrolled in their first TAFE IT course. There were 29 students (46%) studying in online/distance mode, 32 (51%) studying on-campus and the remainder studying in mixed-mode.

3.4 Living Arrangements

Participants were asked to indicate their current living arrangements, by selecting from a list of options. The results are given below in Figure 3. More than one choice could apply to each student (for example, living with a partner and living with dependents), so percentages do not total 100%. Most students live with a partner, dependants, or with other family, with only 11% living alone.

Proportionally more females were living alone, and more females than males lived with dependents. More males lived with other family - for example they lived with their parents or siblings.

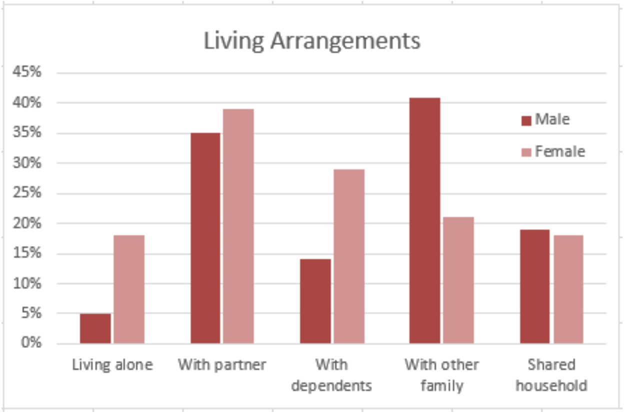


Figure 3: Living arrangements of male/female students.

The living arrangements of the women who live alone or with dependents without a partner or other family could adversely impact on the level of support these students receive during study, in contrast to the male students who would presumably be supported by family.

Primary Caregiver Status: 11% of males and 25% of females indicated that they were the primary caregiver for between 1 and 4 dependents.

3.5 Employment

Participants were asked if they had paid employment, and if so, whether this was full-time, part-time, casual or if they were self-employed. The differences between men and women were significant (Figure 4). More women had employment, inclusive of full-time, part-time or other (Chi Square: p = 0.01; a significance level of 0.05 is used throughout this paper). Both men and women, if they were employed part-time or casually or self-employed, worked an average of 21 hours per week.

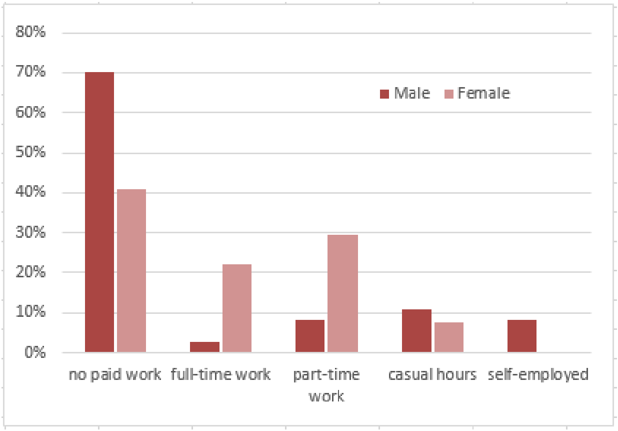


Figure 4: Employment of males and females.

We asked those who had indicated they were not working full-time, whether they were actively seeking employment. As a proportion of those who were not employed full-time, significantly more males (39%) than females (14%) were actively seeking employment whilst studying (Fisher Exact Probability Test: p = 0.04).

Half (50%) of these job-seeking males expected employment (if gained) to impact on their further study, while 66% of job-seeking females expected an impact on their study if they gained employment. This indicates no difference between genders for anticipated impact on further study if employment was gained (Fisher Exact Probability Test: p = 0.54).

3.6 Financial Pressure

To determine the level of financial pressure students may be facing, we asked “If you had to do so in an emergency, could you find $2000 within 48 hours?” This question is included on Australian Bureau of Statistics surveys (for example the ABS General Social Survey - (Australian Bureau of Statistics 2012a)) as a measure of financial stress for individuals and households.

As 60% of female students are working full-time, compared to 30% of males, we may expect that more males than females would answer “no” - that is, more males would be experiencing financial pressure. This was not the case. Around 50% of both males and females identified that they were experiencing financial pressure, despite the higher employment status of female students. The reasons for females experiencing similar levels of financial stress despite higher levels of employment may be associated with female students reporting higher numbers of dependants than male students.

3.7 Kessler 10 (Level of Psychological Distress)

The Kessler 10 (K10) is a standard test of 10 questions designed to identify levels of psychological distress in participants (Kessler & Mroczek 1994) . The test asks participants to indicate how often they have felt various feelings over the past 30 days. Each measure is then scored, with “none of the time” = 1, “a little of the time” = 2, “some of the time” = 3, “most of the time” = 4 and “all of the time” = 5. For the 10 questions, a minimum score of 10 and maximum score of 50 is possible (Andrews & Slade 2001). Scores are then grouped into four levels of psychological distress (Table 3).

K10 score	Categorisation
10 to 15	Low distress
16 to 21	Moderate distress
22 to 29	High distress
30 to 50	Very high distress

Table 3: ABS K10 categories (Australian Bureau of Statistics 2012b)

Not all of the participants chose to answer this section of the survey. The results of the 35 males and 24 females who answered this question are shown below in Figure 5.

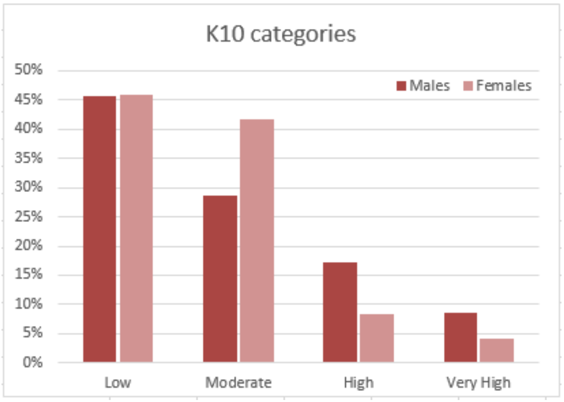


Figure 5: Male and Female K10 scores

There was no difference in psychological distress scores between males and females (Mann-Whitney U Test: p = 0.3192, U_A = 389, z = 0.47).

Of particular interest in these results is that for both males and females, over half of the participants reported at least a moderate level of psychological distress. More research is needed to determine whether this is a phenomenon particular to IT students, or older IT students, or whether TAFE students in general are experiencing significant levels of stress.

3.8 Previous Level of Education

Students were asked to identify the highest level of education they had completed prior to their current course. The results of this question for males and females are shown in Figure 6.

Significantly more females than males had previous tertiary qualifications, either at TAFE or University (Chi-Square test, p = 0.016).

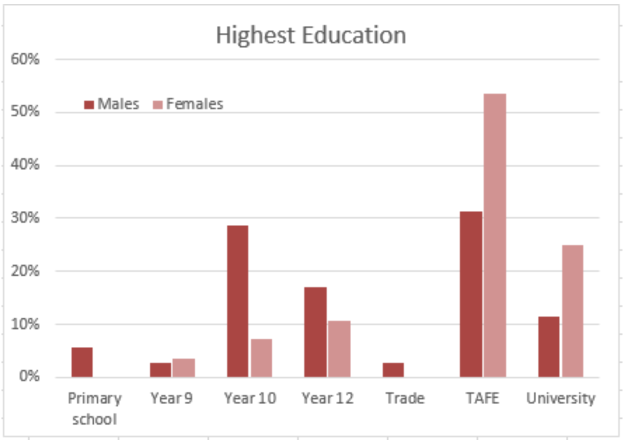


Figure 6: Highest level of previous education

3.9 Reasons for choosing to study in this course

It was proposed that as few females were choosing to continue with higher level TAFE courses, that perhaps their reasons for studying in IT courses in the first place were that their preferred course was not available, or that they were pushed into the course by close friends, parents, or to satisfy other external agents such as being a requirement for continuation of receiving unemployment benefits.

To examine the possible differences in reasons for choice of course, we provided a list of common reasons and asked participants to indicate any that applied to their reasoning for studying their current course. Participants could choose more than one reason. A range of reasons were given, and space was also given for other reasons. The results are displayed for males and females in Figure 7.

There were no significant differences between males and females for any of these reasons (based upon Fisher Exact Probability Tests). It is notable that the most prevalent reason for both males and females was “I have always been personally interested in IT” with about two thirds of all students indicating this as a reason. Other reasons offered for undertaking the course included a desire for formal recognition of existing skills, changing industry and needing re-skilling, and recognition that IT skills were broadly applicable to a wide range of careers.

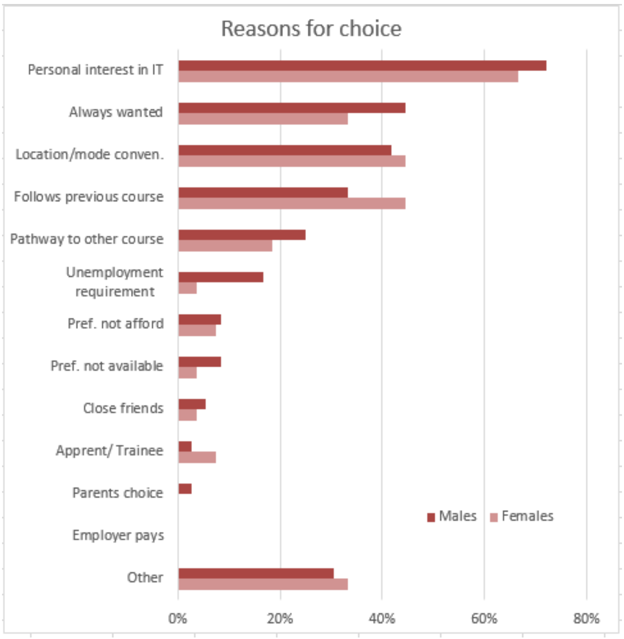


Figure 7: Reasons for choosing current course - males and females.

3.10 Computing Use and Skills

3.10.1 Computer Literacy

Students were asked about their current skill level in using a computer as a tool. Values ranged from 1 ("new to using computers") to 5 ("can use computers for advanced tasks and to format professional documents"). Both males and females generally had high computer literacy (Figure 8), and there was no difference between genders (Mann Whitney U Test, p = 0.4562).

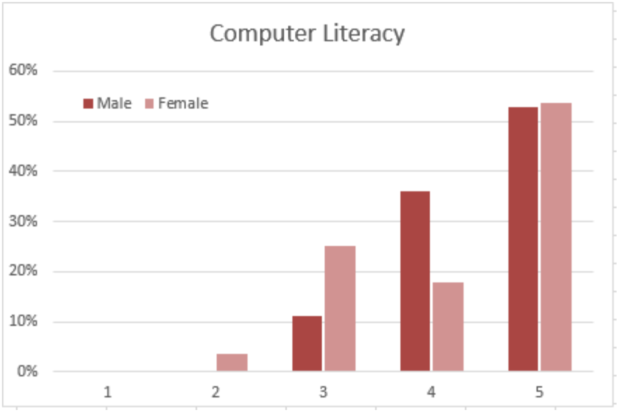


Figure 8: Computer literacy of male and female participants (1 = low, 5 = high).

3.10.2 Computer Use

Time spent on the computer ("computer use") has long been correlated with computing self-efficacy and positive attitudes towards computers (Gardner et al. 1993; Levin & Gordon 1989). The survey participants were asked how many hours (approximately) they spent on a computer at home and (for those who were employed) at work in a week. Our definition of computer use included sending emails, socialising, playing games, studying and other activities.

There was a significant difference between the amount of computer use at home between males and females ((Mann-Whitney U Test: p = 0.0256, UA = 359.5, z = 1.95) with males using computers for longer times (Figure 9), but no difference between genders in the amount of computer use at work (Mann-Whitney U Test: p = 0.18, UA = 66.5, z = 0.9) (Figure 10). There were, however, more females employed than males (Section 3.5) and if each person's total time per week spent on the computer is taken into account, then there is a trend towards females having more total time on the computer each week (Mann-Whitney U Test: p = 0.062 UA = 618.5, Z = 1.54).

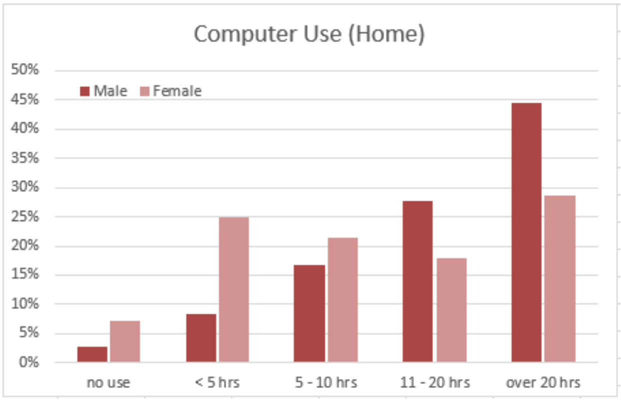


Figure 9: Computer Use at Home - Male and Female

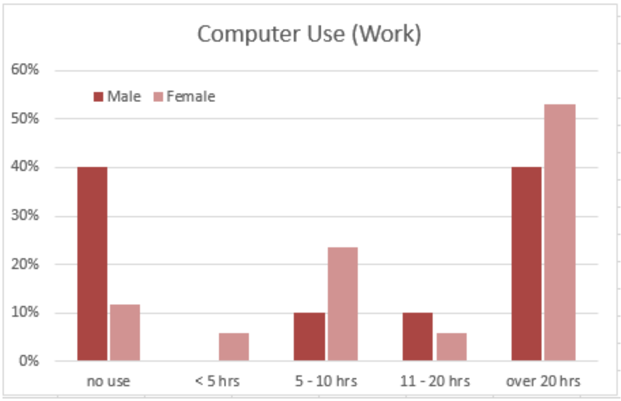


Figure 10: Computer Use at Work - Male and Female

3.11 Attitudes towards the course

Self-efficacy in computing and in study has previously been related to gender (Huffman et al. 2013; Saleem et al. 2011). We asked students to indicate their agreement on a five point Likert scale with the following statements:

- "I feel confident in my ability to study in this course."
- "I feel confident in my ability in information technology."
- "I feel excited about studying in my course."
- "I feel fearful about what might be expected of me in my course."
- "I feel confident that the skills I will learn will benefit me in the future."

3.11.1 Self-efficacy in study

With respect to agreement with the statement “I feel confident in my ability to study in this course”, most students were reasonably confident in their ability to study (Table 4 and Figure 11). There was, however, a significant difference between males and females, with females reporting lower self-confidence (Mann-Whitney U Test - p = 0.0197, U = 296, z = 2.06).

	Median	Mode	Min	Max
Male	4.5	5	2	5
Female	4	4	1	5
All	4	4	1	5

Table 4: Self-efficacy in study ability (1 = low, 5 = high)

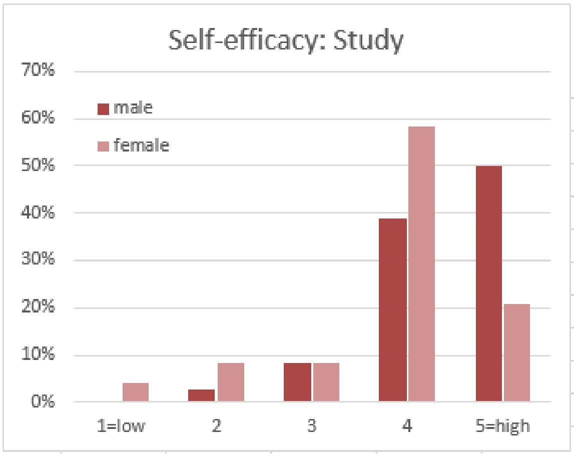


Figure 11: Self-efficacy in study in the course

3.11.2 Self-efficacy in IT

Most students were relatively confident in their abilities in IT as well (Table 5 and Figure 12), as presented by agreement with “I feel confident in my ability in information technology”. There was a trend towards females reporting lower self-confidence, compared to males (Mann-Whitney U Test: p = 0.0808, U_A = 339, z = 1.4).

	Median	Mode	Min	Max
Male	5	5	2	5
Female	4	4	3	5
All	4	4	4	4

Table 5: Self-efficacy in IT (1 = low, 5 = high)

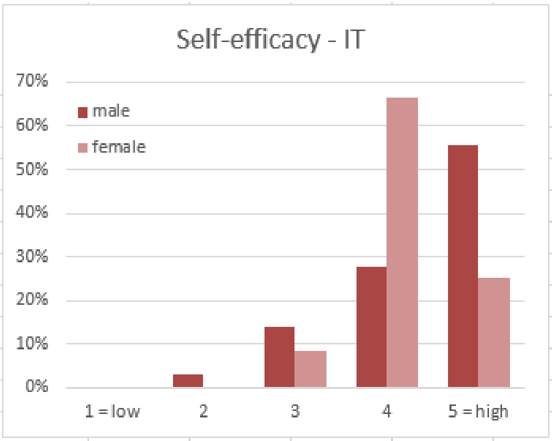


Figure 12: Self-efficacy in IT

3.11.3 General attitude towards course

There was no difference between males and females in their general feelings of excitement about studying in their course (Mann Whitney U Test: p = 0.2148, U_A = 379, z = 0.79) - Table 6.

	Median	Mode	Min	Max
Male	4	5	2	5
Female	4	4	1	5
All	4	4	1	5

Table 6: Excitement about studying: male and female (5 = high)

3.11.4 Fear about expectations

There was no difference between males and females in their level of fear about what might be expected of them in the course (Table 7) (Mann Whitney U Test: p = 0.3192, U_A = 463.5, z = -0.47).

	Median	Mode	Min	Max
Male	3	3	1	5
Female	3	4	1	5
All	3	3	1	5

Table 7: Fear about expectations (5 = high)

While there was no difference between genders, it should be noted that some students were very fearful about what might be expected of them, and most students had some level of apprehension.

3.11.5 Confidence in the course outcomes

There was no difference between males and females in confidence that the skills acquired in their course would benefit them in the future (Mann Whitney U Test: p = 0.1587, U_A = 381.5, z = 1). Most participants rate highly on this measure, indicating generally positive views regarding their study, but there were also participants from both genders who rated this as “low” (Table 8).

	Median	Mode	Min	Max
Male	4.5	5	1	5
Female	4	4	1	5
All	4	5	1	5

Table 8: Perceived benefits (5 = high)

3.12 Future Intentions

Participants were asked what they intended to do when they completed their current course. Participants could choose any number of the provided options - seeking employment in IT, seeking employment in another field, study further in IT, study further in another field, or “other reason”. The results of this question are shown in Figure 13.

Proportionally more women than men were considering employment or study in a field other than IT, while all but 1 male intended to work and/or study in IT. This is clearer when looking at each participant and determining whether they were only interested in studying or working in IT (“IT only”), only interested in studying and working in another field (“Other only”), or if they were more open and considering both ("both"). The results are in Figure 14 below.

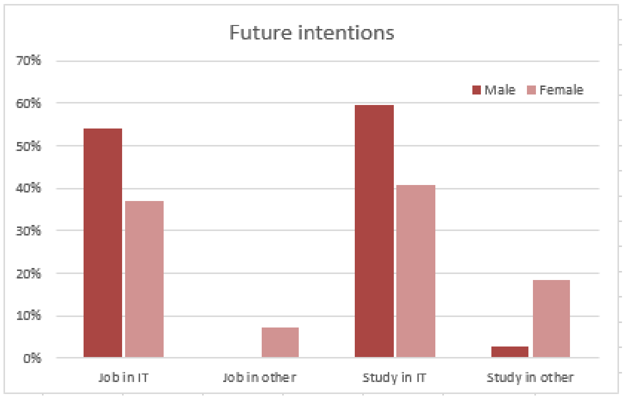


Figure 13: Future Intentions

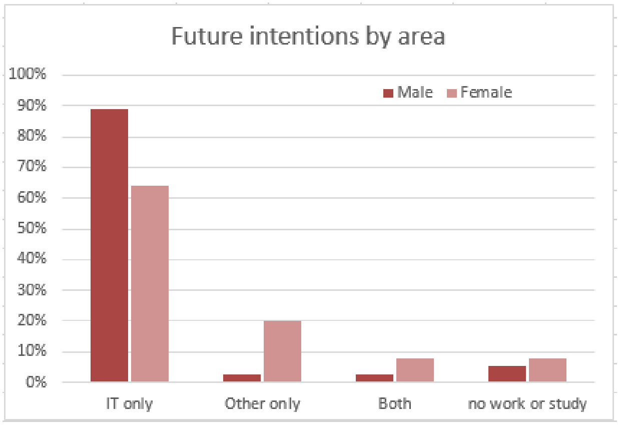


Figure 14: Future intentions by area (IT or other area).

Significantly more males than females (Fisher Exact Test: p = 0.022) were considering study or work in only IT, and had excluded all other possibilities. More females than males (Fisher Exact Probability Test: p = 0.0198) considered working or studying in another field, whether or not they also considered IT. In fact, 20% of females intended to study or work in another field, and had excluded IT as a suitable target for work or further study.

4 Discussion

For the Certificate 1 in IT offered by TAFE NSW - North Coast Institute there are as many female students as male. There are no apparent difference in completion rates for courses between men and women at NCTAFE, yet the proportion of women continuing with further studies in IT declines compared to males. The female students are choosing to *not* pursue further studies in IT at a greater rate than their male counterparts.

This study reports on an initial stage of research being undertaken with the objective of determining the reasons for the gender difference in continuation rates. Although the participants responding to this survey form a non-representational sample biased towards older students compared to the general NCTAFE student population, and biased towards "later courses" of Certificate 4 and Diploma compared to early courses of Certificate 1 and Certificate 2, there are some differences in gender present that may represent differences in the general NCTAFE student population.

Despite holding higher levels of previous education and higher rates of current employment, each indicating a history of relative success, commencing female IT students reported lower levels of self-confidence than men in their capacity to study in their current course. At this stage we do not know whether this lack of confidence is due to an innate characteristic of females, discrimination (as suggested in Valenti 2014), or some other factor.

Women also reported a broader options-horizon than their male counterparts with respect to work opportunities beyond the area of IT. Of all the men studying IT in this study, only one indicated consideration of future study or work in an area other than IT, whereas seven (28%) of females indicated considerations of future employment or study outside of the IT discipline.

The fact that 20% of females indicated that they now *exclude* the possibility of future study or employment in IT is telling, and consistent with the backdrop to this study. Women begin their studies in IT with personal interest rates in the area that are similar to males. Female students are successful in their studies, with completion rates that are similar to males yet, for some reason, they perceive that they lack skills to study in their course, and, for whatever reason, come to the decision that their future career aspirations lie somewhere other than IT.

The IT discipline contains a gender bias demonstrating higher male participation and employment rates. Policies and programs intended to respond to this by intervening and supporting female students, for their life and activities both within and beyond educational institutions, need to better understand the reasons, rationales and perceptions that divide male and female students regarding study and career aspirations in IT. Further investigation in this area is warranted, and is currently in progress.

5 Further Work

The current study has reported upon a participant pool that is skewed towards older students, and towards Certificate 4 and Diploma level courses compared to earlier entry courses. Future iterations of this study will seek a more representational sample with respect to age of students and course of current study.

A series of interviews is underway with some of the participants who completed the survey currently reported. These interviews seek deeper insight into some of the personal narratives, as case studies, with the intent of better understanding the dynamics involved in the decision making process regarding choice of further study, or employment, in IT and other content domains.

As observed by Abraham Wald (Samaniego & Francisco 1984) with respect to the location requirements for placement of armour on warplanes during World War 2, it is not the location of hits upon planes that survive and return that indicate the needed location for armour placement, but the absence of planes with certain areas demonstrating damage, as it is these areas, that indicate catastrophic vulnerabilities. Of the females participating in the current study 41% are undertaking their first course at NCTAFE, and so are presenting information regarding their first experiences and perceptions of study in this area. Further research and analysis is required to drill into the experiences and perceptions of women who *fail* to return to study IT, rather than those who do.

It is important to note that harvesting participants through open invitations to current students, by definition, has excluded those past students who have already chosen, and acted upon, their intentions to not pursue further studies in IT. A strategy will need to be devised by which such people may be identified, and whose participation in completing both questionnaires and interviews, may be acquired, to enable determination as to the reasons why these students have become "missing in action".

6 Acknowledgements

The authors would like to thank the staff and students of the TAFE NSW North Coast Institute for their help and cooperation in this project.

7 References

Andrews, G. & Slade, T., 2001. Interpreting scores on the Kessler Psychological Distress Scale (K10). *Australian and New Zealand Journal of Public Health*, 25(6), pp.494–497.

Australian Bureau of Statistics, 2012a. 4159.0.55.002 - General Social Survey: User Guide, Australia, 2010. *Australian Bureau of Statistics*. Available at: http://www.abs.gov.au/ausstats/abs@.nsf/mf/4159.0.55.002 [Accessed December 10, 2013].

Australian Bureau of Statistics, 2012b. Use of the Kessler Psychological Distress Scale in ABS Health Surveys, Australia, 2007-08. *Australian Bureau of Statistics*. Available at: http://www.abs.gov.au/ausstats/abs@.nsf/Lookup/4817.0.55.001Chapter92007-08 [Accessed December 10, 2013].

Australian Computer Society, 2011. Australian ICT Statistical Compendium 2011. Available at: http://www.acs.org.au/2011compendium/ [Accessed February 16, 2012].

Cohoon, J.M., 2001. Toward improving female retention in the computer science major. *Communications of the ACM*, 44(5), pp.108–114.

Gardner, D.G., Dukes, R.L. & Discenza, R., 1993. Computer use, self-confidence, and attitudes: A causal analysis. *Computers in Human Behavior*, 9(4), pp.427–440.

Hellou, A., 2013. Personal communication.

Huffman, A.H., Whetten, J. & Huffman, W.H., 2013. Using technology in higher education: The influence of gender roles on technology self-efficacy. *Computers in Human Behavior*, 29(4), pp.1779–1786.

Katz, S. et al., 2006. Gender, achievement, and persistence in an undergraduate computer science program. *ACM SIGMIS Database*, 37(4), pp.42–57.

Kessler, R. & Mroczek, D., 1994. Final versions of our non-specific psychological distress scale.

Levin, T. & Gordon, C., 1989. Effect of Gender and Computer Experience on Attitudes Toward Computers. *Journal of Educational Computing Research*, 5(1), pp.69–88.

Logan, K. & Crump, B., 2007. Managing NZ Women in IT. In P. Yoong & S. Huff, eds. *Managing IT Professionals in the Internet Age*. Hershey, PA: Idea Group Publishing, pp. 1–17.

NSW Dept of Education and Communities, 2012. *TAFE NSW Enrolments: Student Profile (2007-2011)*, NSW TAFE. Available at: https://www.det.nsw.edu.au/media/downloads/about-us/statistics-and-research/tafe-nsw-statistics-newsletters/student-profile-2011.pdf [Accessed August 18, 2014].

Roberts, M., McGill, T. & Hyland, P., 2012. Attrition from Australian ICT degrees: why women leave. In *ACE '12 Proceedings of the Fourteenth Australasian Computing Education Conference*. Melbourne, Australia: Australian Computer Society, Inc., pp. 15–24.

Saleem, H., Beaudry, A. & Croteau, A.-M., 2011. Antecedents of computer self-efficacy: A study of the role of personality traits and gender. *Computers in Human Behavior*, 27(5), pp.1922–1936.

Samaniego, M.M. & Francisco, J., 1984. Wald's Work on Aircraft Survivability. *Journal of the American Statistical Association*, 79.386, pp.259–267.

TAFE NSW, 2013. TAFE NSW Statistical Compendium 2012. Technical Paper available from https://www.tafensw.edu.au/about/assets/pdf/TAFE-NSW-Statistical-Compendium-2012.pdf [Accessed October 29, 2014].

Valenti, J., 2014. The female "confidence gap" is a sham. *The Guardian*. Available at: http://www.theguardian.com/commentisfree/2014/apr/23/female-confidence-gap-katty-kay-claire-shipman [Accessed August 20, 2014].

West, M. & Ross, S., 2002. Retaining females in computer science: a new look at a persistent problem. *Journal of Computing Sciences in Colleges*, 17(5), pp.1–7.

Designing a Modern IT Curriculum: Including Information Analytics as a Core Knowledge Area

Magnus Westerlund
Department of Business Management and Analytics
Arcada University of Applied Sciences
Jan-Magnus Janssonin aukio 1
00560 Helsinki, Finland
magnus.westerlund@arcada.fi

Göran Pulkkis
Department of Business Management and Analytics
Arcada University of Applied Sciences
Jan-Magnus Janssonin aukio 1
00560 Helsinki, Finland
goran.pulkkis@arcada.fi

Abstract

Much has happened in the field of Information Technology since 2008 when ACM published its curriculum recommendation for a four year Undergraduate Degree Program in Information Technology. We show an alternative path reflecting what we consider presently requested by the industry and students alike. In this paper we look at the topics from a holistic point of view, not just as traditional machine learning Computer Science courses. We make an argument for widening the scope from machine learning theory, towards analytical service development. We give our proposal of a refined IT curriculum that can be used by other institutions for refining their curriculums.

Keywords:

IT2008, Model Curriculum, Information Technology Education, Information Analytics

1. Introduction

Technology development in the IT sector has during the previous decade, to a large extent, focused on software service development. This has allowed the industry to open up previously closed systems toward information sharing modules. However, during recent years we have seen a new trend emerge, which seems to become the main driver for the IT field in the foreseeable future. In their search for improving customer offerings and increasing productivity, companies and other organizations today turn towards analysing data and information by using advanced machine learning models. The development and usage of these advanced model types have previously mostly been part of master and doctoral Computer Science studies, but we argue that this is about to change. Currently many cloud service providers are developing mainstream offerings of machine learning services that can rapidly be implemented in any software service offering. In consequence, this will require a different skill set compared to what most IT engineers have been taught up till today.

The field of Information Technology undergoes rapid change and requires teaching organizations to continually redevelop their curricula to reflect the changes in the field. Current jobs of professionals with an undergraduate degree in Information Technology (IT, we refer henceforth to the academic discipline and not to the field) are quite often more closely related to the business side of an organization than the jobs of Computer Science (CS) professionals. CS professionals use their scientific competence to solve technical problems and to design software, devices, and systems. An IT professional may, in addition to working on similar tasks, need to understand and communicate sometimes complex dependencies or abstractions between CS professionals' scope and business oriented clients.

With the advent of Information Analytics and software services in general (Software as a Service, SaaS), we see some new trends in the role of IT professionals in the future technology landscape. In the future an IT engineer must also be able to communicate technical implementations, related to information analysis, to business stakeholders. The volume of data and information is growing rapidly in the operational environment of business organizations and other organizations. IT engineers must in the future also have Big Data processing competences such as Information Analytics, which requires profound machine learning skills. We define Information Analytics as a broader knowledge area than Business Analytics that primarily considers business information. Information Analytics deals with all types of information and focuses on the creation of software solutions and services that process this information.

The remainder of this paper is structured as follows. Section 2 contains a literature review of related curriculum design research. Section 3 defines the knowledge areas and learning methods of a proposed new IT curriculum. Section 4 describes the competence requirements of the modules in the proposed new IT curriculum. Concluding remarks are expressed in section 5. Details of the proposed new IT curriculum are shown in an Appendix.

2. Related Curriculum Design Research

Current curriculum design research is usually related to competencies, which the students should achieve. The following five characteristics

1. inclusive and integrative,
2. combinatorial,
3. developmental,
4. contextual, and
5. evolutionary

are used in (Tardif 2006) to describe the features of competency.

In Chang (2014) a method called Q-technique is proposed to obtain the competence requirements of IT enterprises to serve as a basis for developing an IT curriculum:

> "*the purpose of the Q-technique is to consult leading experts within the IT industry, to obtain statements and priorities, to form the universal requirements of IT professional competency.*"

As a conclusion is stated that

> "*IT competencies are structured along the dimensions of information ability, fault tolerance, execution ability, problem solving, learning ability, and innovation ability.*"

A process workbook for implementing competence based education has been prepared for the Clinical and Translational Science Institute of the University of Pittsburgh (Dilmore, Moore, and Bjork 2011). Design of a curriculum in a chosen discipline consists of 13 generic process steps and 3 generic implementation steps.

Design of a competency based curriculum content framework of mechanical technology education is presented in (Sudsomboon 2007). The framework is a set of requirements for knowledge and understanding, for skills, and for attitudes.

ACM currently gives curriculum recommendations for Computer Science (CS), Computer Engineering (CE), Information Systems (IS), and Software Engineering (SE), in addition to the recommendation for Information Technology (IT) (ACM 2008). Examples on using the ACM IT2008 recommendation (Lunt, et al. 2008) for IT curriculum design are found in (Koohang, Riley, Smith, and Floyd 2010, Adegbehingbe and Obono 2012).

Only minor changes were made in the ACM Computer Science curriculum recommendation for 2008 in comparison with the recommendation for 2001, while the changes are significant in the recent recommendation for 2013 (Computer 2013) in comparison with the recommendation for 2008. Required study time in the hardware related knowledge areas 'Architecture and Organization' and 'Networking and Communication' has been cut more than 50% and two new knowledge areas, 'Information Assurance and Security' and 'Parallel and Distributed Computing' have been added. In 'Architecture and Organization' there is a stronger emphasis on multi-core parallelism and virtual machine support, and in 'Networking and Communication' there is increased attention to wireless networking. The required study time in 'Information Assurance and Security' is more than 20% of the total required study time when the distribution of this knowledge area in other knowledge areas is taken in consideration.

The significant changes in the Computer Science curriculum recommendation for 2013 (Computer 2013), in comparison with the recommendation for 2008, strongly motivate large changes in the ACM IT2008 Model Curricula recommendation in (Lunt, et al. 2008). ACM Education Board has actually in 2012 established an exploratory Review Task Group for Information Technology (RTGIT) to review the IT2008 curriculum recommendations (Paterson et al. 2013). In Zilora et al. (2013) a new curriculum proposal for teaching IT and also the addition of analytics as an overarching theme for the curriculum is presented.

3. Knowledge Areas and Learning Methods of a new IT Curriculum

When developing our proposed curriculum we primarily make our recommendations for change based on the experiences in research within generative information infrastructures (Henfridsson and Bygstad 2013), the three branches of analytics (descriptive, predictive, and prescriptive analytics, outlined for business analytics in Delen and Demirkan (2013), but also applies generally for information analytics), natural computing (Shiffman 2012), software engineering, and modern pedagogy of integrating natural sciences and programming through an Active Learning methodology.

3.1 Knowledge Areas

We started our curriculum development by considering whether we should merely add one or two more pillars to the existing five pillars of IT, presented in the IT curriculum recommendation in Lunt, et al. (2008). The IT field has however changed tremendously over the past six years. By just adding more pillars we feel that the curriculum would become too general and would therefore be insufficient for IT students and for industry requirements on future IT experts. We concluded that a clear focus is required in order to ensure the necessary depth in the education of IT engineering experts. We propose that an IT professional should acquire competences in four different knowledge areas, which also reinforce each other, see Figure 1. However, we should point out that the current broader competence, based on the ACM IT2008 Model Curricula (Lunt, et al. 2008), is still relevant today. With the introduction of cloud computing services like PaaS (Platform as a Service), SaaS (Software as a Service) and IaaS (Infrastructure as a Service) over the past years, we conclude that in the future there will be fewer jobs requiring hardware knowledge and to some extent hardware related networking skills. Most new IT jobs will require software skills related to IT services and scalability issues. A good example of this development is the Amazon AWS Elastic Beanstalk (AWS 2014) solution for implementing Java applications to a publicly available application server without any in-depth hardware knowledge. We believe that it is therefore essential to look towards the future and to do our best as educators to anticipate coming industry needs.

As the IT curriculum proposal in Zilora et al. (2013), our IT curriculum proposal also has a focus on analytics. It consists of the following 8 modules:

- General Studies
- 4 Core technical modules
    - a basic study module on Web and Visualization
    - professional study modules in Analytical Methods and Data Science, Service Oriented

Architectures and System Design, and Machine Learning and Decision Support Development

- an extension study module in Business Processes
- Practical Training
- Thesis Work

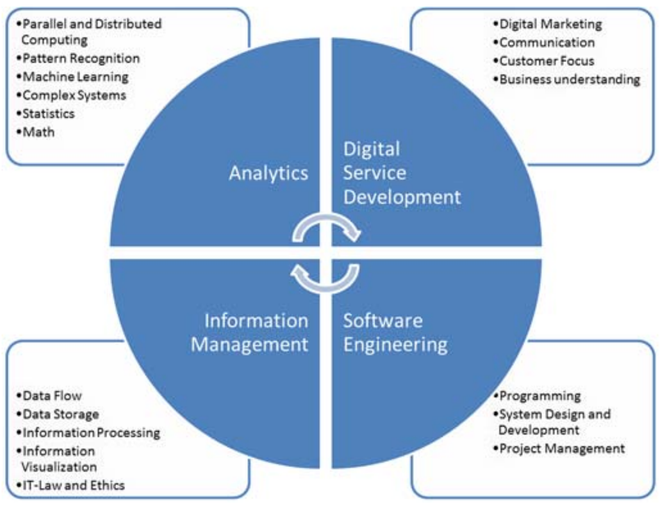


Figure 1. Knowledge areas for an IT professional.

Each module corresponds to 30 ECTS (European Credit Transfer and Accumulation System) (European 2009) and is further divided into course units with a minimum size of 5 ECTS. The learning outcomes and achieved competences from each module are discussed in Section 4 and the course units in each module are listed in the Appendix.

We take the position in this paper that IT should have a clearer focus towards being the glue between the four other ACM disciplines (CS, CE, IS, SE), for which curriculum recommendations are given (ACM 2008), and a quantitative analyst. IT should focus equally on both Information and Technology. The former, Information, is represented in our knowledge areas as Information Management and Digital Service Development, see Figure 1. The Technology part is characterized by Software Engineering and Analytics. We find the current ACM pillars in Lunt, et al. (2008) too confined as such, e.g. the web systems pillar indicates only one type of system, suggesting that mobile or desktop are not as important. The learning outcomes from the Human Computer Interaction (HCI) pillar in Lunt, et al. (2008) we find imperative, but we rather see it from a more general scientific viewpoint of information visualization than as a separate pillar. Therefore HCI does not exist as a knowledge area in Figure 1, as it encompasses all knowledge areas in Figure 1.

3.2 Learning Natural Sciences in Programming Course Units

Natural sciences and programming are integrated using an Active Learning methodology. The three often referenced Active Learning methods, Collaborative, Cooperative, and Problem-based, are deployed. Active Learning in engineering studies has been shown to improve learning results significantly (Prince 2004).

The main focus for all natural science topics throughout the degree is to provide students with a fundamental mathematical understanding of machine learning and data science relevant concepts. We realize this through the perspective of natural computing significant course units. This, we think, assists the student in the learning processes by providing a reference model that the student can relate to, by seeing how nature functions.

During the first year, students will study natural science course units for a total of 15 ECTS. The studies focus on applied mathematics, statistics, physics, and introduce them to mathematical programming from the onset. To give one example, physics will be taught in the form of game programming in order to help the student to visualize essential concepts. The intention is to positively reinforce the students' learning process, to focus the students on their own experience and development. We, as in most Western European societies, have noticed a decline in natural science ability and interest among the recent generation of students. We hypothesize that students will become more motivated if the focus in natural sciences is not only on abstract topics, but also involves creative and responsive elements.

Throughout the second year students learn to understand the concept of particle systems, as a collection of independent but interactive objects (Reeves 1983). The students should be able to implement a system of particles interacting based on forces, motion, waves, and oscillations, in order to understand the notion of variations over time. Concepts such as amplitude, frequency, period, degrees and radians and their transformations become familiar, e.g. in programming a pendulum example (Shiffman 2012). An important part is matrix calculations, including scalar and matrix operations, transverse matrix, inverse matrix, determinant and solution of matrix equations.

4. Competence Requirements for an IT Engineering Curriculum, Focusing on both Information and Technology

In this section we start by describing the general competences we have found to be essential for the future IT engineer. This is followed by an examination of the core technical competences required to handle the diverse responsibilities. During this examination we do not limit ourselves to Information Analytics, but rather present a holistic view of the core technical competences that were identified. In essence we define the future competence need to be based on equal parts of Information and Technology studies. The level of competence depth gained for the degree, i.e. learning outcomes, should follow the European Qualifications Framework (European 2014) level 6. Level 6 competences are defined in the context of responsibility and autonomy as " *Manage complex technical or professional activities or projects, taking responsibility for decision making in unpredictable work or study contexts; take responsibility for managing professional development of individuals and groups.*"

4.1 General Competences

As the SaaS model has become an important driver for digital entrepreneurship and business growth, we believe students need an in-depth understanding of creating such services. Service innovation is mostly a customer driven process so customer involvement is important (Hanseth et al. 2012, Alam and Perry 2002). Understanding and being able to anticipate the latent needs of customers is a complex task, but research shows that customer involvement is often crucial and leads to the development of more innovative services, regarding both originality and user value (Matthing et al. 2004). Therefore it stands to reason that communicational, social, and business skills among IT engineers are of great importance. We consider that this is becoming an even more prominent feature of successful IT professionals in the future. In Lunt, et al. (2008) it is stated that IT "*focuses on preparing graduates who are concerned with issues related to advocating for users and meeting their needs within an organizational and societal context through the selection, creation, application, integration and administration of computing technologies*". In our IT curriculum proposal the studies will therefore contain a significant amount of general competences in topics such as communication and social interaction.

In Delen and Demirkan (2013) the need for business analytics is defined as "*At a time when firms in many industries offer similar products and use comparable technologies, business processes are among the last remaining points of differentiation.*" The field of analytics allows the companies to extract "*every last drop of value from those processes*". This requires at least a basic understanding of how a company functions and how different business processes can be measured. We therefore defined an elective module as business processes, were students can get an introduction into functionality of companies. We introduce three different core processes: marketing, logistics and financial management.

We also offer an alternative to this module, an entrepreneurship focused module for those students that prefer founding their own companies.

4.2 Overview of Core Technical Competences

In designing the curriculum the outcome goals for the degree were defined as that graduates have competences to analyse information and are able to develop software services for the digital world, here without focusing on any specific context area. The student should learn to plan and construct software for web, mobile, and cloud services or applications. The student should also have the ability to visualize, analyse, and handle data that exist in various forms. The final technical goal that was formulated was that the student should be able to motivate the use of different types of machine learning models in order to get answers from various hypotheses or to questions based on processed data.

We acknowledge that Big Data has become an important impetus for many technology oriented and customer driven companies. However, we find the analytical understanding from an academic perspective to be the fundamental driver for implementing new services. In our view the size of data refers more to a tool proficiency skill than to a pure competence. Scalability and parallelization as technological competences should explain the phenomenon of Big Data. Although the argument from a mathematical point of view often requires a separation of small and big data, as e.g. the sample size differ (small n=10´s; big n=all). We consider, however, the hypothesis creation to be part of a quantitative analyst's (data scientist's) job description, rather than the IT engineer's.

The current ACM IT curriculum recommendation highlights information assurance and security as comprehending all pillars (Lunt, et al. 2008). These recommendations for information assurance and security are relevant also in our IT curriculum proposal, with the addition of cloud service security and analytics for implementation of security services.

4.3 Web and Visualization Module

The initial technical module is intended to teach students the structure of information and interactive programming. We assume that the students, when they start, have fundamental skills in handling a computer and common software. Our previous experience is that new students often have limited understanding of how to divide a problem into its essential sub-components and how to re-assemble the solutions of the sub-components into a structured result. Therefore, we have set the learning outcome for the first year studies to achieve proficiency in web development platforms and programming languages. The students should be able to develop web applications and explain the web architecture, in order to demonstrate their problem solving ability. The student should also be able to produce visually appealing and easy to use user interfaces, which includes responsive design i.e. that the layout changes depending on which client the visitor uses. Our intention is to teach both imperative and declarative programming from the start in order to support the student's learning experience in understanding both information structure and interactivity. Research has shown that visual perception and thinking are linked through an intrinsic relationship (Arnheim 1980). Therefore teaching information visualization through the use of descriptive programming should support an understanding of both spatial and temporal relationships. These relationships can be directly related to imperative programming constructs, and should arguably support the student in forming the initial mental pictures of abstract constructs, help them to reflect on their practice, and inform them about future designs (Walny 2011).

4.4 Analytical Methods and Data Science Module

The task of handling data is for an IT engineer likely to become a more important competence than before. The current ACM curriculum recommendation has Databases as one of its pillars (Lunt, et al. 2008). The shift in technology we are currently experiencing, requires us to broaden the perspective from databases (the process of storing and extracting data) to include analytical and visual methods for dealing with data, and also to understand technically very different types of storing/processing methods in scalable architectures. With the introduction of Big Data tools such as Hadoop we have gone from mostly focusing on efficient data structures to designing efficient algorithms that process seemingly unstructured data. Hadoop (Welcome to Apache Hadoop 2014, Borthakur 2007) can be described as a distributed database that allows users to process information directly on the node were the data resides. Therefore understanding the concept of parallelization for solving problems becomes essential.

Before a dataset can be processed in a machine learning model it often requires extensive pre-processing, also called feature engineering. There are several stages that are part of pre-processing, e.g. checking data for validity, coding, dealing with missing values, normalization, and feature extraction, to name a few. The pre-processing stage is usually considered to be the most time consuming and important stage in terms of improving the end result. In addition to understanding the earlier mentioned stages it also requires object oriented programming skills in order to be able to automate data pre-processing. Once a machine learning model has given an output, this data needs to be post-processed to create decisions and often some type of visualization is needed for a human to understand the output. (Baesens 2014)

Taking this into account we have defined the targeted competences for the module to be that the student can manage, organize and visualize data. The student should also be able to justify how the data should be stored to comply with technical, legal and contractual provisions, but also be able to evaluate security risks in data management and apply data security in computer networks. Regarding programming competences the student should be able to plan and produce secure applications based on object oriented programming. Students learn to develop essential sequentially coded algorithms and also to implement these algorithms with parallelized code. Examples of such algorithms are various sort/process tasks, which in later modules can be used to explain more complex programming methods, e.g. the MapReduce programming model (Dean and Ghemawat 2008).

4.5 Service-Oriented Architectures and System Design Module

As the software industry has matured over the past two decades it has meant that we currently emphasize architectural design more than ever. The focal point for this development has been the Service Oriented Approach which represents a baseline for a distributed architecture with no direct reference to implementation (Erl 2004). The distributed architecture defines well-formed access points through Web services, which when made public, open up the information infrastructure to become a shared, evolving, and open experience (Hanseth 2002). Henfridsson and Bygstad (2013) identified three generative mechanisms at the core of creating successful information infrastructures: innovation, adoption, and scaling. These were considered self-reinforcing processes that spawn new recombinations of resources. As user adoption increases, more resources are invested and therefore the usefulness of the infrastructure increases. True service scaling attracts new partners by offering incentives for collaboration.

We consider it important that students understand that creating successful software requires a much broader understanding than only a programming understanding. Hence, we will devote a large portion of the third year towards raising awareness and understanding of how software architectures can be made scalable by utilizing cloud infrastructures and software defined networks (Sommerville 2013). As a basis for an innovative infrastructure we will focus the attention on how students obtain a critical understanding of descriptive data mining or text analytics. From a technical point of view the student must be able to defend the chosen architecture by referring to established software patterns.

The objective for the third year is to give an introduction to machine learning models. The core focus is on autonomous agents, evolutionary algorithms, and statistical pattern learning (text analytics). Students learn to design ranking algorithms that allow them to implement objective functions for various optimization problems. Earlier studies have focused on the individual particle, but during the third year "herd behaviour" is introduced as to appreciate how the agents' own decisions influence the group and vice versa.

As is mentioned in section 4.1, the recommended general competences include topics such as marketing and digital marketing to enhance the students' understanding of consumers and service adoption.

4.6 Machine Learning and Decision Support Systems Development Module

In Davenport (2013) it was claimed that we are currently embarking on the third evolution of analytical services. Analytics 1.0 was the era of Business Intelligence, 2.0 the era of Big Data, and 3.0 is the era of data-enriched offerings. This new era requires new types of technologies, but also uses many of the open source tools, e.g. Hadoop, or cloud computing services developed for the previous eras (Davenport 2014). Therefore we claim that the new IT engineering challenge will be to combine various tools and services with appropriate models and data sources, to deliver new insights to the end user.

Thus, the fourth and final technical module focuses on Service Oriented Decision Support System (SODSS) development. These service types are often offered as distributed collaboration components, produced by many partners, and consumed by end users for decision making. Examining the SODSS environment as a process three major service classifications emerge: data, information, and analytics. Data-as-a-Service (DaaS) allows any business process to access data wherever it resides. The technical implementation is often performed through Master Data Management (MDM) and/or Customer Data Integration (CDI). Information-as-a-Service (IaaS) typically refers to a refinement of data and to making information available quickly to people, preferably in real-time. This opens up technical challenges such as real-time data formatting, in-memory computations, and parallel transaction and event processing. Analytics-as-a-Service (AaaS) tends to focus on insights drawn from machine learning models. These models can be of a descriptive nature, but are often focusing on predictive and prescriptive elements. AaaS consumes information services in order to deliver different types of Enterprise Analytics or other end user relevant analytics. Technical issues include scaling, interface dependencies, in-memory computing, dealing with machine learning models as black boxes, and system stability. (Demirkan and Delen 2013)

During the fourth and final year students gain an ability to utilize more advanced machine learning models for both predictive and prescriptive analysis. They will learn to appreciate the inner workings of various learning methods and solve non-linear problems. The main focus of the studies will be on the computing side, i.e. that students learn how to create analytical systems. They will learn essential heuristic techniques and their mathematical explanation for problem solving, learning, and discovery.

Consequently the student should learn to predict events based on prior data through the use of machine learning. Machine learning models we refer to here are defined through the universal approximation theorem stating that any arbitrary continuous function can be estimated. This is done through a non-linear mapping of the input vector into a high-dimensional feature space, which in turn is connected to an output layer. (Haykin 2013) Students should thus learn to deal with high-dimensionality problems that allow them to master the implementation of advanced models for solving both regression and classification problems. This also requires the student to have a fundamental understanding of optimization techniques that can be used to demonstrate an optimal solution to a given problem.

Once students have an empirical understanding of dealing with machine learning models, focus is shifted towards creating services such as decision support systems and automated expert systems.

4.7 Thesis Module

In the thesis module the student learns to manage projects and understands how development is executed in agile projects. Students can express themselves in their native language both orally and in writing, as required by regulation. The student will be able to write a publication that summarizes the development of a project in a scientific manner.

5. Conclusions

The Information Technology field is rapidly developing and at the same time changing other fields it comes into contact with, everything from healthcare to the automotive industry. Rometty (2013) (CEO of IBM) commented the current technological shift towards analytical services as "*…this is a thirty to fifty year, long-term project, which requires the next generation of computers, i.e. the self-learning computer*". During this time we will likely see the IT curriculum change many times and curricula for new disciplines develop, e.g. currently Business Analytics degree programs have hastily been developed at many Business Schools. The coming changes to the ACM IT curriculum recommendation should take into consideration Information Analytics in order for IT degree programs to stay relevant in the future. By tackling machine learning through development of analytical services and not through mathematics as is often done in Computer Science, we believe the area can be opened up to a greater engineering audience than before.

6. References

ACM: ACM Curricula Recommendations. http://www.acm.org/education/curricula-recommendations. Accessed 19 Aug 2014.

Adegbehingbe, O.D. and Eyono Obono, S.D. (2012): A Framework for Designing Information Technology Programmes using ACM/IEEE Curriculum Guidelines. *Proc. World Congress on Engineering and Computer Science WCECS 2012* I.

Alam, I. and Perry, C. (2002): A customer-oriented new service development process. *Journal of Services Marketing* 16(6):515-534.

Arnheim, R. (1980). A plea for visual thinking. *Critical Inquiry* **6**(3):489-497.

AWS: AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/. Accessed 19 Aug 2014.

Baesens, B. (2014): *Analytics in a Big Data World: The Essential Guide to Data Science and Its Applications*. John Wiley & Sons.

Borthakur, D. (2007): The Hadoop Distributed File System: Architecture and Design, The Apache Software Foundation. http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf. Accessed 19 Aug 2014.

Chang, C.-C. (2014): Obtaining IT Competencies for Curricular Development using Q-technique. *International Journal of Academic Research in Business and Social Sciences* 4(3):60-74.

Computer Science Curricula. (2013): The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) IEEE Computer Society. http://www.acm.org/education/CS2013-final-report.pdf. Accessed 19 Aug 2014.

Davenport, T.H. (2013): Analytics 3.0. *Harvard Business Review* 91(12), 64-72.

Davenport, T.H. (2014): *Big Data at Work: Dispelling the Myths, Uncovering the Opportunities*. Harvard Business Review Press.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM* **51**(1):107-113.

Delen, D. and Demirkan, H. (2013): Data, Information and Analytics as Services. *Decision Support Systems* **55**(1):359-363.

Demirkan, H. and Delen, D. (2013): Leveraging the capabilities of service-oriented decision support systems: Putting analytics and big data in cloud. *Decision Support Systems* 55(1):412-421.

Dilmore, T.C., Moore, D.W. and Bjork, Z.J. (2011): Implementing Competency-Based Education. A Process Workbook 2009-2011prepared for the Clinical and Translational Science Institute in University of Pittsburgh.

http://www.academic.pitt.edu/assessment/pdf/CompetencyBasedEducation.pdf. Accessed 19 Aug 2014.

Erl, T. (2004): *Service-oriented architecture: a field guide to integrating XML and web services*. Prentice Hall PTR.

European Commission: Descriptors defining levels in the European Qualifications Framework (EQF). http://ec.europa.eu/ploteus/content/descriptors-page. Accessed 19 Aug 2014.

European Communities (2009): ECTS Users' Guide. http://ec.europa.eu/education/tools/docs/ects-guide_en.pdf. Accessed 19 Aug 2014.

Hanseth, O. (2002): From systems and tools to networks and infrastructures – from design to cultivation. Toward a theory of ICT solutions and its design methodology implications. http://heim.ifi.uio.no/~oleha/Publications/ib_ISR_3rd_resubm2.html. Accessed 19 Aug 2014.

Hanseth, O., Bygstad, B., Ellingsen, G., Johannessen, L. K. and Larsen, E. (2012): ICT Standardization Strategies and Service Innovation in Health Care. *Proc. International Conference on Information Systems*, Orlando, USA.

Haykin, S. (2009): *Neural Networks and Learning Machines*, 3rd ed. USA, Pearson Prentice Hall.

Henfridsson, O. and Bygstad, B. (2013): The Generative Mechanisms of Digital Infrastructure Evolution. *Management Information Systems Quarterly* **37**(3):896-931.

Koohang, A., Riley, L., Smith, T. and Floyd, K. (2010). Design of an Information Technology Undergraduate Program to Produce IT Versatilists. *Journal of Information Technology Education* 9:99-113

Lunt, B.M., Ekstrom, J.J., Gorka, S., Hislop, G., Kamali, R., Lawson, E., LeBlanc, R., Miller, J. and Reichgelt, H. (2008): Information Technology 2008 Curriculum Guidelines for Undergraduate Degree Programs in Information Technology. http://www.acm.org/education/curricula/IT2008%20Curriculum.pdf. Accessed 18 Aug 2014.

Matthing, J., Sandén, B. and Edvardsson, B. (2004): New service development: learning from and with customers. *International Journal of Service Industry Management* 15(5):479 – 498.

Paterson, B., Granger, M., Impagliazzo, J., Sobiesk, E., Stockman, M. and Ming Zhang, M. (2013): Should IT2008 be revised? *Proc. 14th annual ACM SIGITE conference on Information technology education*, New York, USA, 53-54, ACM Press.

Prince, M. (2004): Does active learning work? A review of the research. *Journal of engineering education* 93(3):223-231.

Reeves, W.T. (1983): Particle systems—a technique for modeling a class of fuzzy objects. *ACM SIGGRAPH Computer Graphics* 17(3):359-375.

Rometty, V. (2013): A Conversation with Ginni Rometty. http://www.youtube.com/watch?v=SUoCHC-i7_o. Accessed 19 Aug 2014.

Shiffman, D. (2012): The nature of code:[simulating natural systems with processing]. Selbstverl.

Sommerville, I. (2013): Teaching cloud computing: a software engineering perspective. *Journal of Systems and Software* 86(9): 2330-2332.

Sudsomboon, W. (2007): Construction of a Competency-Based Curriculum Content Framework for Mechanical Technology Education Program on Automotive Technology Subjects. *Proc. ICASE Asian Symposium.* Pattaya, Thailand.

Tardif, J. (2006): L'évaluation des compétences : Documenter le parcours de développement. Montréal, QE: Chenelière Éducation

Walny, J., Carpendale, S., Riche, N.H., Venolia, G. and Fawcett, P. (2011): Visual thinking in action: Visualizations as used on whiteboards. *IEEE Transactions on Visualization and Computer Graphics* **17**(12):2508-2517.

Welcome to Apache Hadoop. Retrieved on 6/6/2014 from http://hadoop.apache.org/. Accessed 19 Aug 2014.

Zilora, S.J., Daniel Bogaard, D. S. and Jim Leone, J. (2013): The changing face of information technology. *Proc. 14th Annual ACM SIGITE Conference on Information Technology Education*. New York, USA, 29-34, ACM Press.

Appendix

Details of our proposed IT curriculum for an Undergraduate Degree Program are shown in Figure 2 and Table 1. Figure 2 illustrates the curriculum structure at our university. We define the course units to be covered for each of the modules (with the exception for practical training) in Table 1.

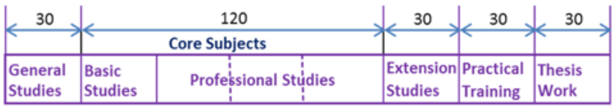


Figure 2. Structure of proposed IT Curriculum.

Module Title	Level	Course Units Covered
General	General	Arcada 360; Introduction to Academic studies; Communications and Public Speaking; Nature of Code 1 - Introduction to Mathematical Programming; Statistics and Probability; Second Language
Web and Visualization	Basic	Web Development; Front-end Programming; Back-end Programming; Web services, Databases and CMS; Computer Architecture and Operating Systems; Nature of Code 2 - Vectors and Forces
Analytical Methods and Data Science	Professional	Information Visualization; Data Structures and Algorithms; IT-Law and Ethics; Concurrent Programming; Nature of Code 3 - Oscillation and Particle Systems; Network Protocols and Security;
Service Oriented Architectures and System Design	Professional	Network Communication and Cloud Technologies; Nature of Code 4 - Autonomous Agents and Cellular Automata; Nature of Code 5 - Fractals and Evolution of Code; Descriptive Analytics - Data/Text Mining; Software Defined Networks; Analysis and Design, UML and Design Patterns
Machine Learning and Decision Support System Development	Professional	Analytical System Design; Image and Speech Recognition Algorithms; Decision Support System Development and Verification; Predictive Analytics - Neural Networks; Prescriptive Analytics – Optimization; Process Optimization
Business Processes	Extension	Introduction to Business Administration; Introduction to Marketing; Digital Advertising; Introduction to Logistics; Intercultural Business; Introduction to Financial Management
Thesis Work	Thesis	Project Management; Academic Writing; Research Methodology and Seminar; Thesis Work (15 ECTS)

Table 1. Course Units Covered

Quality Assurance using International Curricula and Employer Feedback

Marta Lárusdóttir
Reykjavik University
Reykjavik, Iceland
marta@ru.is

Mats Daniels
Uppsala University
Uppsala, Sweden
mats.daniels@it.uu.se

Roger McDermott
Robert Gordon University
Aberdeen, Scotland
roger.mcdermott@rgu.ac.uk

Abstract

The focus of this paper is the quality assurance process for the bachelor program in the School of Computer Science at Reykjavik University, which is a combination of outcome- and process-oriented quality assurance. Faculty members and employers of graduates provided information for the quality assessment. The results provide both detailed quantitative data and more qualitative information that give all stakeholders a variety of ways to interpret the status of the quality of education. This type of assessment has raised the awareness of the faculty members on how abstract topics and learning outcomes from an international standard can be used when revising the curricula of a particular course. A notable feature of this type of analysis is its use of employer-generated data to examine graduate knowledge and skills. The contribution of the paper is to provide an example of how a quality assurance process can be made more valuable to both faculty and degree stakeholders by combining outcome- and process-oriented quality assurance strategies.

Keywords: Quality assurance, Evaluation, Degree programs

1 Introduction

Quality assurance of education programs is a complex task and can serve several different functions such as helping to identify pedagogical strengths and weaknesses in a program, or, in extreme cases, providing evidence for its cessation. This complexity is compounded by the fact that the process itself can be conducted by different stakeholders, e.g. national agencies or individual departments within a particular institution. Moreover, the methodology used - specifically the focus of the quality assurance process and the type of assurance procedures used - may significantly affect the conclusions that are drawn. In most cases attention is directed to either the features of the educational experience (including curriculum content, course administration, delivery and assessment mechanisms...) or to an assessment of the abilities of the graduating students. In both these cases, fundamental questions arise about what precisely should be measured and which set of criteria should be used. These issues are even more problematic when attempting to assess areas for which there may be no obvious or well-established metrics, e.g. professional skills such as intercultural competence. Furthermore, consideration also needs to be given to whether the issues to be measured are known in advanced by those being evaluated, since this could potentially lead to "cosmetic" adjustments made to subvert the accuracy of the evaluation process.

The focus of this paper is the quality assurance process taking place in the computer science bachelor program at Reykjavik University, Iceland. The process was partly influenced by the Swedish national quality assurance process for computer science programs performed in 2012/2013. The Reykjavik process is of interest in that it combines an assessment of program content and delivery with evaluation of graduates' abilities. Rationale for choices, methods for conducting the quality assurance, some results as well as conclusions will be covered in this paper. We highlight two key features of the Reykjavik process. The first is the use of the ACM/IEEE computer science curricula 2013 (ACM/IEEE 2013) (henceforth referred to as the "ACM Curricula 2013") to bridge the gap between the typically fairly abstract national degree criteria and the more tangible aspects of course implementation, and to provide a rather concrete description for evaluating findings. The second is the use of employer responses to assess relevant graduate attributes.

2 Quality Assurance

As stated above, quality assurance is a complex endeavour in which the details of context are important. In this current work, the academic department is taken to be the main stakeholder and performs the quality assurance process in order to ascertain strengths and weaknesses so as to improve the program. There are several ways to ensure the validity of this kind of process. One is to base any review on the accreditation criteria for computer science programs (ABET 2010) devised by internationally recognised accreditation organisations such as ABET (formerly known as the Accreditation Board for Engineering and Technology). ABET conducts assessments, including site visits, outside the US and have also influenced national quality assurance programmes, e.g. in Estonia. Another effort to ensure validity is conducted by the European association for quality assurance in higher education (ENQA) (ENQA 2013), which is an association within the European Union evaluating quality assurance processes in its member countries.

This kind of quality benchmarking is useful, especially for so-called process-oriented quality assurance which focuses on what an education program contains and how it is delivered. An alternative strategy for conducting the quality assurance process is outcome-based assurance where the abilities of students after a course or a degree program are assessed, and this has recently become more popular. ABET changed their assessment strategy towards this at the turn of the century (Lattuca et al. 2006) and Sweden is at the end of four year national quality assurance cycle for all degree programs which mainly uses outcome-based procedures (HSV 2012). Process-oriented assurance focuses on the general process by which education is carried out and there are many readily available sources of information which may be used to feed in to this analysis. However, there is often a lack of attention to the experience of the learner. By contrast, outcome-based assurance tries to assess the quality of the program by determining if suitable outcomes have been achieved. This lends itself to a student-focused approach but assumes that there is agreement on what outcomes should be measured and what constitute the criteria for success. Since both alternatives have their strengths and concomitant weaknesses, there is current interest in looking at approaches which use positive aspects of both practices to evaluate the quality of a program. One such attempt is that of Reykjavik University Computer Science department.

3 The Reykjavik University Setting

3.1 The Computer Science Program

The bachelor program in computer science at Reykjavik University started in 1998 and the taught content was, at that time, strongly influenced by the 1991 version of the ACM/IEEE computer science curriculum (Tucker 1991). The program had an extensive review in 2008 based on the 2001 version for the computer science subfield (ACM/IEEE 2001). During this overhaul, the revision of the standard from 2008 was also taken into consideration (ACM/IEEE 2008). The program includes 17 mandatory course units in computer science and mathematics for a total of 102 ETCS (one ETCS is 1/60 of a "student year") and a mandatory final group work project that is 12 ECTS for each student. In addition, students can select between four "emphasis lines" which consist of 30 ECTS in courses related to their focus subject.

3.2 Quality Assurance Method

In 2013, the program was the subject of a quality assurance evaluation as part of an ongoing national cycle of Higher Education review based on the Quality Enhancement Framework (Rannis 2011). The main aim of this framework is to support the quality assurance efforts of Icelandic Higher Education institutions by providing guidance on the objectives, requirements and operational procedures for evaluating quality at both the institutional and departmental level. In terms of compliance with QA regulation, the main source of documentation, the Quality Enhancement Handbook for Icelandic Higher Education, specifies that "all institutions will be required to conduct regular internal reviews covering each of their subject areas" and the subject-level review was scheduled for the School of Computer Science within the 2013 calendar year.

An important question for such reviews is the basis on which the quality assurance process should progress. As mentioned in section 2 in this paper, there are two basic approaches generally termed process-oriented and outcomes-oriented. The former tends to examine the structural elements of the educational process (e.g. content, curriculum, learning objectives, teaching styles) and map it against some set of trans-institutional standards which act as a benchmark for best practice in the area. The second approach looks at the output of the educational process and tries to determine whether the students that have undergone the experience do indeed possess the knowledge, skills (and attitudes to learning) that the program seeks to deliver. A number of difficulties present themselves in this situation. For example, a choice needs to be made on what constitutes an appropriate criteria of success, how the assessment of these measures should take place, and who provides the data for making such a decision. One influential input to the discussions for the Reykjavik review was the recent (2012/2013) national quality assurance process for computer science in Sweden, which took a strongly outcome-based approach.

The obvious starting point for any examination of educational quality in an Icelandic degree program is the national degree criteria (Rannis 2011). Unfortunately, while providing a useful framework to discuss general aspects of learning at the subject level, these criteria were found to be too abstract to serve directly as the basis for constructing learning objectives for the various course units. Following historical precedent, therefore, it was decided to use the 2013 Ironman draft of the ACM Computer Science curriculum as a bridging document linking the high-level pedagogical objectives of the national criteria to specific learning objectives within particular course units (ACM/IEEE 2013).

An attempt was made to map the general objectives of the national degree criteria to the more specific statement of skills contained in chapter 3 of the ACM/IEEE curriculum document. For example, it was possible to map the statement from the national criteria that a student graduating from a bachelor of science program should be "capable of interpreting and presenting scientific issues and research findings", (Education ministry 2011) to the ACM Curriculum guidelines on communication and organizational skills: "Graduates should have the ability to make effective presentations to a range of audiences about technical problems and their solutions. This may involve face-to-face, written, or electronic communication. They should be prepared to work effectively as members of teams. Graduates should be able to manage their own learning and development, including managing time, priorities, and progress." [ACM/IEEE 2013, p.22].

The example given above illustrates two things. Firstly the ACM document articulated a description of the various knowledge and skills elements to be found within the generic computer science curriculum areas at a much finer level of granularity than the national document itself and this enabled clearer discussion of the criteria for success. Secondly the ACM document served a normative function by acting as a benchmark for comparing the disposition of knowledge and skill elements within the courses of the Reykjavik program with those that the ACM curriculum deemed to be necessary elements of a computer science bachelor program. This gives the process-oriented element of the quality assurance process but it does not address the problem of how to assess outcome-based criteria such as the ability to demonstrate appropriate capabilities in a graduate working environment. In order to evaluate this aspect of the program, information on the performance of newly-graduated students was sought from employers.

3.3 The ACM/IEEE Curricula

The ACM/IEEE document identifies two main pedagogical elements of the curriculum: *Knowledge areas* and *Characteristics of graduates*. The former specifies the content areas of the subject whereas the latter identifies the more general, interdisciplinary skills and competencies that a student should develop through engagement with the educational program.

3.3.1 Knowledge Areas

The knowledge areas are part of the "Body of knowledge" section of the curricular document (ACM/IEEE 2013) and define the topical areas of computer science as seen by ACM and IEEE. There are 18 knowledge areas in the 2013 standard, see table 1 in section 5.1, and two of them are new to this version, i.e. "Information assurance and security" and "Parallel and distributed computing".

Each knowledge area is described by a list of sub-areas with associated topics and learning outcomes, and the document also specifies a number of "curricular hours" assigned to each sub-area. The sub-areas are identified as either "core" or "elective" and the core parts are in their turn subdivided into "tier-1" and "tier-2", each with an associated number of "curricular hours". This classification builds on a view that all computer science programs should ensure that all of the tier-1 and most (preferably 90-100%, but at least 80%) of the tier-2 is mastered by all their students. A complete computer science program should also offer a significant part of the elective material.

3.3.2 Characteristics of Computer Science Graduates

The characteristics of computer science graduates define the competencies these students should have at graduation. The idea behind these definitions is to capture overarching characteristics that typically span several of the knowledge areas and which are important for graduate success in the computer science profession. There are eleven characteristics identified in the ACM curricula 2013 (ACM/IEEE 2013), see table 2 in section 5.2. The expectation is that at least an elementary level of all should be achieved at graduation by all students.

4 Analysis of Educational Setting and Delivery

The analysis of the educational setting and delivery is done in two parts, a process-oriented part and an outcome oriented part. The process-oriented part of the quality assurance evaluation at University A was targeted on the educational setting, both on the course content and the course learning outcomes. The learning outcomes for course units are also relevant for the outcome-oriented part of the evaluation, but to a lesser degree since the evaluation of these are focused more on the competencies students have gained at the time of completing the course and not on the holistic competences graduates have when completing the bachelor degree.

The educational setting is analyzed by two separate methods. The first was to compare how many of the topics and learning outcomes suggested in the ACM curricula 2013 were situated in mandatory courses at Reykjavik University. All faculty member teaching mandatory courses took part in this evaluation (n=10), including 2 professors, 2 associate professors and 6 assistant professors. Each faculty members checked how many of the topics suggested in the ACM standard are covered in their course and the degree to which the learning outcomes articulated in the course unit documentation matched that found in the ACM document. This comparison was structured by the knowledge areas from the ACM curricula 2013.

The second method was to estimate how much focus was placed on each of the characteristics of computer science graduates in the mandatory courses. A guideline document was developed to assist the faculty members in conducting this comparison. There was an initial workshop for the faculty when the quality assurance evaluation was introduced and the chosen process explained. The process involved several stages in order to guide faculty members in how to conclude their part of the assessment and was concluded with a joint workshop analyzing the results from both methods.

4.1 Analysing Educational Setting - Coverage of Topics and Learning Outcomes

For the first method of analysing the educational setting, a spreadsheet with topics and learning outcomes for tier-1 and tier-2 of the knowledge areas was composed. The faculty were asked to fill in the coverage for each topic and learning outcome associated with the knowledge areas related to the courses they teach using the guideline document.

The spreadsheet contained the topics for each knowledge area in each course and the extent to which the topics were covered. The coverage of the learning outcomes for each knowledge area was captured in terms of the ACM/IEEE levels of achievement (termed familiarity, usage, or assessment) as well as describing assessment method, i.e. (written) exam, oral (exam), group (project), (individual) assignment, and other. All of this was also subdivided into tier-1 and tier-2.

In the analysis phase, the coverage for each knowledge area was computed as a percentage and the level of learning outcome was compared to the expected level in the ACM curricula 2013.

4.2 Analysing Educational Setting - Emphasis on Characteristics of Graduates

A spreadsheet with the mandatory courses and the specified characteristics of computer science graduates (with the exception of the first, which was assumed to be covered by the analysis of knowledge areas covered) was composed. Faculty were then asked to fill in the level at which each characteristic is supported. This was encoded as a "0", "1" or "2", i.e. not covered or only marginally mentioned (0), part of the course (1), and central to the course (2). In addition, faculty were asked to comment on their evaluation.

In the analysis phase, the number of 0's, 1's and 2's were computed for each mandatory course. The number of courses with 0's, 1's and 2's for each characteristic was also computed.

4.3 Analysing Educational Delivery – The Employers' Assessment of Graduate Skills

The main target of the second part of the evaluation was employer perception of graduate skills. Ten companies were chosen and semi-structured interviews were conducted, each lasting for about one hour. The interviewees work in different domains: two at big software companies (more than 100 employees), two at middle size companies (around 40 employees), two at web development companies, two at telecommunication companies, one at a software development department in a bank and one at a gaming company. Three of the interviewees were female and seven males. Typical roles of the interviewees were: Director of the company, director of IT department and chief development officer so they all had a managerial role and had been involved in hiring people for the last three to 13 years. All except one had hired graduates from SCS at RU, and the percentage of hirings from RU was typically 50-70% of all the hirings.

The interviews were all conducted at the workplace of the interviewees, typically in a meeting room. Two faculty members from SCS at RU attended each interview and, roughly speaking, one of them led the interview whereas the other one took notes. The interviews lasted from 45 minutes up to one hour. The interviews were semi-structured and the major topics covered in the interviews were background information about hirings and the company, their opinion of graduates from SCS at RU, their comparison of graduates from RU to graduates from other universities and their thoughts about possible new study programs or courses. Near the end of the interview we asked the interviewees if they had some general comments or questions. All interviews were audio recorded for further references. Interviewees were asked to fill in a web based questionnaire based on the characteristics of graduates described in the ACM Curricula 2013 (ACM/IEEE 2013) that was sent to them after the interview.

4.3.1 Interviews with Employers

The interviewees were asked about their background at the companies and if they had been involved in hiring graduates from Reykjavik University. They were also asked to provide numbers of hirings of BS graduates in Computer Science from Reykjavik University. The main focus of the interviews was to ask about the employees' opinion of the performance of the graduates from Reykjavik University, and especially to get their views of the strengths and weaknesses of the graduates' education. In addition, interviewees were asked if they thought that some knowledge or skill was missing, and whether there was a need for new courses, or lines of emphasis, which would satisfy their own need to recruit better qualified graduates.

All interviewees were willing to discuss these issues and gave good comments and feedback on these questions.

4.3.2 Questionnaire to Employers

A web-based questionnaire was constructed based on eleven characteristics of computer science graduates from the ACM Curricula 2013. Employers were then asked to rate how well graduates from Reykjavik University performed on each of these, based on a 5 point Likert scale, e.g. employers were asked to rate if they agreed that: "Graduates from Reykjavik University have good project experience skills". They were also asked to rate the importance of each of the characteristics (e.g. "Project experience skills are important for my company").

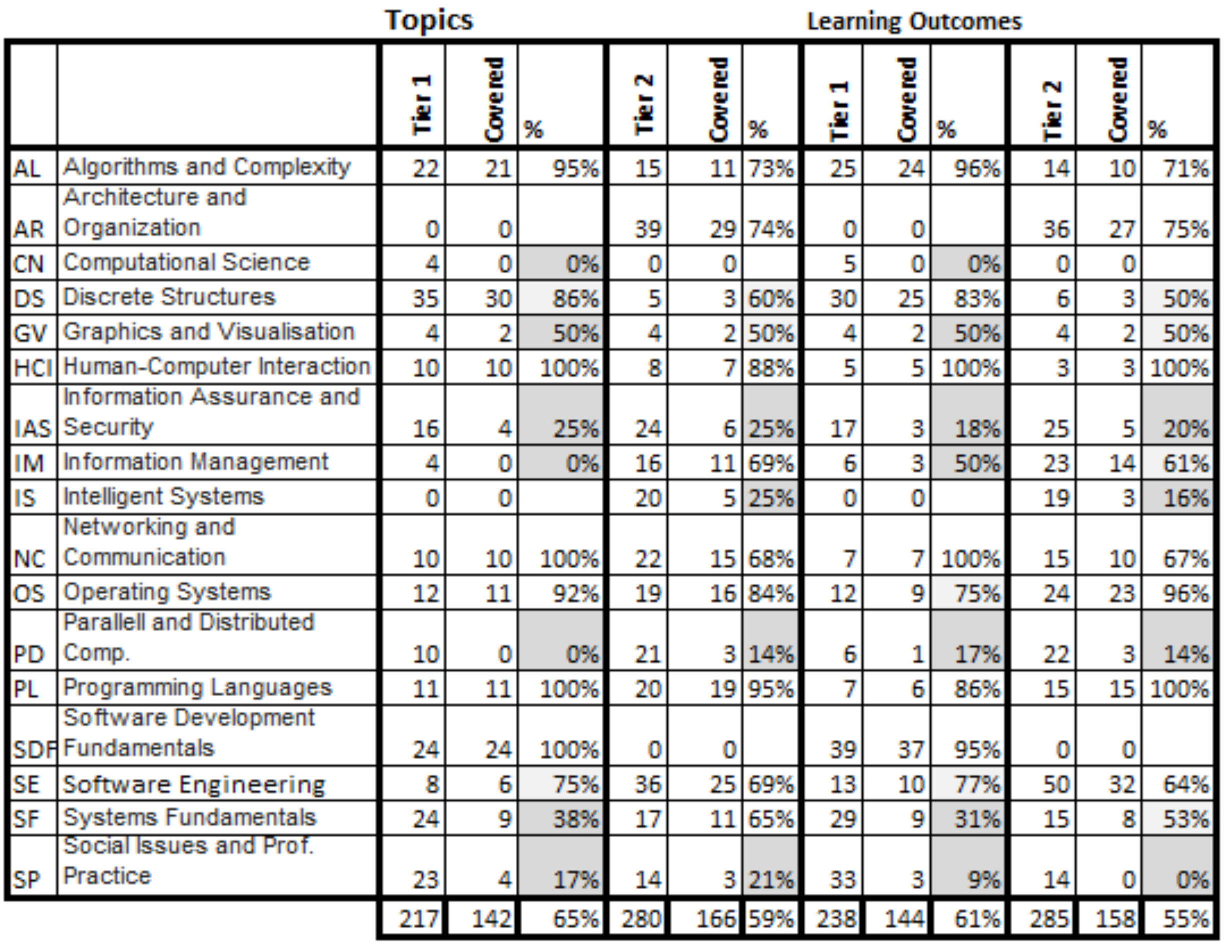
As the data sought by the questionnaire was much more detailed than that provided by the interviews, it was decided to send this afterwards in the expectation that this would maximize the quantity and quality of the data returns. Only seven interviewees concluded the questionnaire. One interviewee had not hired any graduates from RU, so this person was naturally dismissed concluding the survey, but despite several emails, the two missing responses were not forthcoming. The questionnaire was anonymised, so it was impossible to find out which people did not respond.

5 Findings

The results of the analysis of the educational setting are summarized below. Table 1 presents the knowledge area topics and the learning outcomes for those knowledge areas, and table 2 presents the characteristics of computer science graduates. The summary of results from the employer survey is given in table 3.

5.1 Coverage of Knowledge Areas

According to the ACM/IEEE curricula, all of tier-1 should be covered for all computer science programs. Analysis of table 1 regarding the coverage of knowledge areas (KAs) reveals that this is not the case for the mandatory courses at Reykjavik University, which is, perhaps, not surprising since the program was being compared to a cutting edge standard. Coverage of six KAs are fully covered or almost so, meaning that close to half of the tier-1 KAs are satisfied. However six are either not covered at all or only covered to a small extent and three are covered to some degree, which together with a total coverage of 65% of tier-1 indicates a need for change if striving to follow the ACM curricula 2013.



		Topics						Learning Outcomes					
		Tier 1	Covered	%	Tier 2	Covered	%	Tier 1	Covered	%	Tier 2	Covered	%
AL	Algorithms and Complexity	22	21	95%	15	11	73%	25	24	96%	14	10	71%
AR	Architecture and Organization	0	0		39	29	74%	0	0		36	27	75%
CN	Computational Science	4	0	0%	0	0		5	0	0%	0	0	
DS	Discrete Structures	35	30	86%	5	3	60%	30	25	83%	6	3	50%
GV	Graphics and Visualisation	4	2	50%	4	2	50%	4	2	50%	4	2	50%
HCI	Human-Computer Interaction	10	10	100%	8	7	88%	5	5	100%	3	3	100%
IAS	Information Assurance and Security	16	4	25%	24	6	25%	17	3	18%	25	5	20%
IM	Information Management	4	0	0%	16	11	69%	6	3	50%	23	14	61%
IS	Intelligent Systems	0	0		20	5	25%	0	0		19	3	16%
NC	Networking and Communication	10	10	100%	22	15	68%	7	7	100%	15	10	67%
OS	Operating Systems	12	11	92%	19	16	84%	12	9	75%	24	23	96%
PD	Parallell and Distributed Comp.	10	0	0%	21	3	14%	6	1	17%	22	3	14%
PL	Programming Languages	11	11	100%	20	19	95%	7	6	86%	15	15	100%
SDF	Software Development Fundamentals	24	24	100%	0	0		39	37	95%	0	0	
SE	Software Engineering	8	6	75%	36	25	69%	13	10	77%	50	32	64%
SF	Systems Fundamentals	24	9	38%	17	11	65%	29	9	31%	15	8	53%
SP	Social Issues and Prof. Practice	23	4	17%	14	3	21%	33	3	9%	14	0	0%
		217	142	65%	280	166	59%	238	144	61%	285	158	55%

Table 1: Coverage of knowledge areas - topics and learning outcomes

Looking at tier-2, which is recommended to be covered at above 80%, we see that 59% of this is covered and thus does not conform to the ACM benchmark. The KAs covered well at the tier-1 level are also catered for at tier-2 and a few of the KAs not covered at the tier-1 level are covered to a better degree at the tier-2 level. The problematic ones are those deemed to be covered at neither level.

The information assurance and security and the parallel and distributed computation KAs are among these, which is not surprising since these were only introduced in the 2013 version of the curriculum recommendation. The social issues and professional practice topic is the third KA not covered at the required level in either tier-1 or tier-2. This probably reflects the observation that faculty as well as program coordinators have a focus on the technical aspects of computer science. This assumption is further investigated in a forthcoming article (Daniels et al 2015). Two other KAs worth noting are intelligent systems and computational science, both of which are peripheral to the intentions of the program and consequently it is not unexpected that these scores are low.

Some of the areas are covered in elective courses, but this is deemed to not be of interest here, since the intent is to investigate the areas that all students should learn.

The data for learning outcomes show slightly worse results than the preceding investigation of topics covered. The KAs with poor coverage reappear when looking at the learning outcomes, which is perhaps not surprising. The two new areas are just slightly worse with regard to assessing learning objectives, but a significant low score is presented by the social issues and professional practice KA. This KA is barely covered at all when it comes to assessment, which is probably related to faculty being unsure about how to assess such competencies in general. Previous work on assessing professional competencies (Daniels 2011, Cajander et al. 2012) can provide support so as to improve this situation.

5.2 Coverage of Characteristics of Graduates

Investigation of table 2 regarding the focus on characteristics of computer science graduates, called competencies in the following, reveals that just over a third of the mandatory courses cover all of the competencies. However, a more interesting question is whether there are aspects of developing competencies that come up in few courses and at a superficial level, since those cases could indicate a lack of provision for allowing development of the competencies in question.

Characteristics of graduated	Introduction to CS	Software requirements and design	Programming	Practical project	Discrete math I	Discrete math II	Software Engineering	Computer Architecture	Algorithms	Problem solving	Calculus and Statistics	Data structures	Operating systems	Computer communications	Databases	Programming languages	Web programming	Final Project	**Not in the course**	**Covered**	**A core subject in the course**
Familiarity with common themes and principles	1	1	0	1	1	1	1	1	2	0	1	1	1	1	1	0	1	1	3	14	1
Appreciation of the interplay between theory and practice	0	1	0	2	1	1	1	1	2	1	1	1	1	1	1	1	1	1	2	14	2
System-level perspective	1	2	0	2	0	0	1	1	1	0	0	1	2	1	1	1	2	1	5	9	4
Problem solving skills	1	2	1	2	1	1	1	1	2	2	1	1	2	1	0	0	1	2	2	10	6
Project experience	1	1	0	2	0	0	2	0	0	0	0	0	0	0	1	0	2	2	11	3	4
Commitment to life-long learning	2	1	1	1	0	0	1	0	1	0	0	0	0	0	1	1	1	1	8	9	1
Commitment to professional responsibility	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1	11	7	0
Communication and organizational skills	1	1	0	2	0	0	2	0	1	2	0	0	0	0	1	0	1	2	9	5	4
Awareness of the broad applicability of computing	2	1	1	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	8	9	1
Appreciation of domain-specific knowledge	1	1	1	1	0	0	1	1	0	0	0	0	2	1	0	0	0	2	9	7	2
Number of 0	1	0	5	1	7	7	0	4	4	7	7	6	4	4	3	7	1	0			
Number of 1	7	8	5	4	3	3	8	6	3	1	3	4	3	6	7	3	7	6			
Numer of 2	2	2	0	5	0	0	2	0	3	2	0	0	3	0	0	0	2	4			

Table 2: Coverage of characteristics of computer science graduates

None of the faculty members emphasise the "Commitment to professional responsibility" as a core competency in their course. During discussions about this result among faculty members, two alternatives were proposed for improvements. The first was to embed elements of this topic in a variety of course units within the program. The second alternative was to include this material in the course unit called "Introduction to computer science" in their first semester. While the "Project experience" competence may appear to be underrepresented within the program, being only covered in seven courses, in four of these it is the major pedagogical component. Many courses also include project work as a problem solving experience, so students are often developing this competence by working in groups to solve smaller projects.

This type of activity provides a process-oriented analysis of the Reykjavik program and illustrates the benefits that can be gained by comparing the current curriculum with an international standard. However, it does not address the question of how effective such a curriculum is for student-learning. For this, it is more natural to use an analysis which looks at output data, that is done in the next section.

5.3 Outcome Oriented Assessment

For an outcome-oriented assessment of curricular content/knowledge areas, one source of information are the standard, published output measures such as exam result data which can be correlated with a range of comparable programs in similar institutions. However, it is much more difficult to assess outcomes for the specified graduate characteristics in that way. In addition, exam result data and degree classifications do not necessarily give a complete picture of the range of skills and competencies developed by students throughout their period of study; this may only become apparent when they are asked to demonstrate such capabilities over a sustained period within a professional working environment. It is important therefore to examine the views of stakeholders such as employers who can provide a more contextualised analysis of such competencies.

In order to do this, employers were asked to estimate how well newly-qualified graduates from RU fulfilled these characteristics and how important each characteristic is to their company. Responses were given using a Likert scale from 1 to 5, see results in table 3.

Item	Applies to RU graduates	Important for company
Technical understanding	4.00	4.86
Familiarity with common themes and principles	3.71	4.86
Appreciation of the interplay between theory and practice	3.57	4.00
System-level perspective	3.86	4.43
Problem solving skills	3.86	4.86
Project experience	3.86	4.29
Commitment to life-long learning	3.71	4.86
Commitment to professional responsibility	3.43	4.57
Communication and organizational skills	3.86	4.29
Awareness of the broad applicability of computing	3.14	3.71
Appreciation of domain-specific knowledge	4.00	4.29

Table 3: Summary of the results from the employer survey

There were five characteristics that the employers rated as very important to their company, having an average above 4.5 in importance. These were: "Familiarity with common themes and principles", "Problem solving skills", "Commitment to life-long learning", "Commitment to professional responsibility" and "Technical understanding". For the first four of those the difference between the importance rating and how well that competence applies to RU graduates is 1.0 or more (marked in red in the table) indicating that these characteristics should be a particular focus for curriculum development when changing the program in the future.

It should be noted that "Commitment to professional responsibility" was not emphasised as a core subject in any of the mandatory courses in the curriculum, so that particular result is not unexpected. "Familiarity with common themes and principles" and "Commitment to life-long learning" are each only emphasised as a core subject within one course unit, so again, the difference between the needs of employers and graduate performance may not be surprising. However, problem solving skills are emphasised in six compulsory courses, so the difference between the two ratings is disappointing and indicates an important gap for that competence that needs to be addressed through curricular enhancement. In this particular case, further investigation suggested that the difference could be related to some respondents' perception of a recent, local decline in programming skills.

5.4 Further Results from the Employers

The feedback from the employer survey indicated that there were no major concerns about the levels of competence of the RU graduates and in general, the view was positive. Four interviewees mentioned that RU prepares graduates well for working in the industry after their studies, and that RU students were proficient with the tools and processes used in the industry, particularly the agile methodology. Three of the employers had groups working in parallel in other countries (Ukraine, Serbia and Britain), which allowed them to discuss the relative strengths of the RU graduates with those they have worked with from other countries. The respondent working with a team in Ukraine stated that in his/her opinion, the Ukrainian employees are better programmers and want to discuss methods, understand and have opinions on solutions. The respondent having a team in Serbia described that those team members have more theoretical education and not as much practical experience as graduates from RU. Finally, the respondent working with a team in Britain noted that it is harder to get a permanent job in Britain than in Iceland, so the British graduates are more focused and more concerned about doing a good job than employees here in Iceland in his opinion.

When asked about RU graduates weaknesses, there were various answers. Some employers mentioned that RU graduates should develop more professional behaviour and show better discipline in their work. Two respondents mentioned that the programming skills of RU graduates should be improved, and one informant mentioned that RU graduates could have better skills in designing from scratch using design patterns. Two informants mentioned that RU graduates could improve their testing skills and one mentioned in particular that automatic testing should be emphasised more in the RU programs. One respondent mentioned that their company has one tester per every four programmers and it has been hard to find good testers on the market.

When asked about, if there were some courses or topics missing in our curriculum, the answers were really spread, mentioning web programming, front end programming, testing and management of IT systems. The employers were asked specially about the structure of the studies. One employer mentioned that he would like us to have four lines: one for "hard core" programming; one for web programming; one system administration (system administrators are mostly not educated at a university level), and one testing line. Additionally one informant wanted to divide our studies in two lines; one programming line and one front-end programming line.

The employers in general want better work ethics, emphasis on testing and more commitment to quality. It is also important to keep in mind that the employers felt more individual differences between their employees, rather than thinking of them as RU graduates, graduates from other universities in Iceland or abroad. Therefore probably many of their comments can be interpreted as holding for CS graduates in general rather than only for the RU graduates. However, their comments are useful to improve the studies at RU in order to prepare RU graduates better for their future jobs in industry.

6 Discussion

In this section we will first discuss the validity of the findings and then summarise and discuss the lessons learned.

6.1 Validity of the Findings

The validity of the findings is subject for discussion. While some element of confirmation bias will be present due to the evaluation being done by faculty with a vested interest in a good outcome, the classification system for inclusion is fairly transparent and standard moderation practices would mitigate against this. There is also a question of consistency both in terms of how well the faculty entered numbers into the spreadsheets, and more importantly, their understanding of what the terms meant. However, faculty information events prepared academic staff for the process and this would also serve to reduce these kinds of errors.

The evaluation process itself did involve revisiting decisions on the allocation of scores and the concluding session, in which faculty discussed the data provided some degree of confidence in the robustness of decisions about scores and agreement on the meaning of the classification criteria. It should be stressed that the objective of the assessment was to see if there are extensive gaps in the coverage of the knowledge areas suggested in the ACM standard in the curriculum for computer science at Reykjavik University. Consequently the objective was not measure exactly the coverage, but to gather information on whether there were some knowledge areas where the curriculum differed greatly from the topics and learning outcomes suggested by the ACM standard. In the absence of a systematic error, this objective would be reached even if some faculty members were too positive/negative about the details of their own course units.

One further source of concern is the likelihood that the technical aspects of the curriculum are better understood by the faculty involved in the evaluation than those that relate to competencies. That results and the generally poorer outcome for the competencies in the process-oriented analysis, indicate that further work is required to establish a common understanding of what competencies are and how they can be developed and assessed. This point also applies to other stakeholders, such as employers, who appear to be even less accustomed to vocabulary related to competencies than faculty.

6.2 Lessons learned

It was generally felt by faculty that combining the process-oriented evaluation based on the ACM standard with outcome-oriented evaluation, based on interviews with employers of graduates, provided a good methodology for obtaining a more complete picture of the quality of the program. The process generated both detailed quantitative data and more qualitative information that gave stakeholders a good mixture of results to interpret the status of the quality of the education. This assessment has raised the awareness of the faculty members of what topics and learning outcomes should be included in their courses, when revising the curricula of the courses. Already a half a year after the exercise, some of the faculty members have used the results of the assessment to iterate their course content and learning outcomes for the course.

In future iterations of similar comparisons between the topics and learning outcomes in the RU curricula to the ACM standard, it would be beneficial to ask the faculty member responsible for each course to estimate how much of the course is used on topics and learning outcomes that are covered in the standard and then how much time is used on other topics and learning outcomes. This would help to estimate how much is taught beyond a given standard and will therefore give more holistic picture of the curricula. Another lesson is that faculty members were asked to mark how each learning outcome is tested, e.g., individual or group assignment, is it on the test, etc., but that data was not analysed, so that information is not needed in future comparisons.

Conducting the interviews with employers of graduates from RU was a positive experience. All the respondents appeared to be open minded and willing to give feedback, both on the skills of the RU graduates and, in more general terms, on how the CS education could be improved to better satisfy the needs of their company. We asked them to estimate how many employees they had hired from RU the last five years, but unfortunately did not manage to give adequate notice before requiring this information. Our experience was that they would have needed a longer time to answer that question properly. Asking them to fill in a questionnaire after the interviews was good, because the interviews dealt with general issues and so the response to the questionnaire was on a more detailed level. However, it was hard to obtain the data in which we were interested, so one alternative would be to ask the informants to fill in the questionnaire on paper during the interview. The downsides of this alternative are that filling in the questionnaire would take time from the interview itself and it might affect the interviewees' responses by observing them. Additionally filling in the questionnaire during the interview would probably change the focus of the interviewees to talking about the questions they had answered in the questionnaire.

7 Conclusions

Going through a quality assurance process can be quite frustrating and consume a great deal of time and energy. There were much controversy around the Swedish national process especially about the lack of feedback to the degree granting institutions about how to enhance their educational setting as a result of the experience. The Reykjavik process was, on the other hand, received quite positively after some initial complaints about having to go through with the work. It therefore provided an excellent opportunity to discuss the results and move towards improving the computer science program.

The ACM/IEEE computer science curricula 2013 [1] is an important contributor to the positive reaction in Reykjavik. It served well as a replacement for local learning objectives in the computer science program, since those were rather outdated and were instead a target for improvement after the quality assurance process. The good fit of the ACM curricula [1] with the national degree criteria in Iceland [13] was important for those responsible for reporting to the national project.

The Reykjavik quality assurance process illustrates how the ACM curricula 2013 [1] can be used to provide a well-founded base for further discussions about development of an education program. While we believe that there is no clear resolution to the question of how compliant a program should be with regard to the ACM tier-1 and tier-2 criteria or how much conscious deviation from the standard should be allowed, we nevertheless believe that it is of high value to bring it up to the table for discussion.

It is also a welcome finding that the ACM curricula could be used to capture traditionally abstract learning objectives regarding general competencies. The ACM curricula turned out to be an excellent base for conducting semi-structured interviews and constructing a survey in order to get information from employers of students from the education programme. Satisfaction of learning objectives regarding general competencies is in our opinion often quite questionable in computer science programs of today and we hope this work will encourage others to look seriously into how to achieve this.

8 Acknowledgements

We would like to thank professor Anna Ingolfsdottir and assistant professor Henning Ulfarsson for their contribution in the subject level review while planning, analyzing and reporting the findings described here. Furthermore we would like to thank all the faculty members that took part in the review for their contribution.

9 References

ABET (2010) Criteria for accrediting computing programs, ABET, Inc, Baltimore

ACM/IEEE (2013) Computer Science Curricula 2013, Ironman draft, February 2013, http://ai.stanford.edu/users/sahami/CS2013/ironman-draft/cs2013-ironman-v1.0.pdf, assessed December 19, 2013

ACM/IEEE (2001). Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science. Journal of Educational Resources in Computing (JERIC), 1 (3es), Fall 2001. Retrieved from: http://www.acm.org/education/education/education/curric_vols/cc2001.pdf

ACM/IEEE (2008). Joint Task Force on Computing Curricula: Computer Science Curriculum 2008: An Interim Revision of CS 2001. Report from the Interim Review Task Force, Association for Computing Machinery, IEEE Computer Society. http://www.acm.org//education/curricula/ComputerScience2008.pdf

Cajander, Å., Daniels, M., and McDermott, R. (2012) "On Valuing Peers – Theories of Learning and Intercultural Competence", Computer Science Education, vol 22, pp 319-342

Daniels, M. (2011) Developing and Assessing Professional Competencies: a Pipe Dream? Experiences from an Open-Ended Group Project Learning Environment, Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology, nr 808, Acta Universitatis Upsaliensis, Uppsala.

Daniels, M., Cajander, Å., Clear, T., and McDermott, R. (20015) Collaborative Technologies in Global Engineering: New Competencies and Challenges, International Journal of Engineering Education (accepted for publication)

Education ministry (2011) National qualification framework for higher education, http://www.stjornartidindi.is/DocumentActions.aspx?ActionType=Open&documentID=afd35930-4c5a-4de4-bd7e-2134da404446, assessed August 28, 2013

ENQA (2013), Guidelines for external reviews of quality assurance agencies in the European higher education area, ENQA, Occasional papers, 19

HSV (2012) Högskoleverkets system för kvalitetsutvärdering 2011–2014, Rapport 2012:15 R, Högskoleverket, Stockholm, Sweden

Icelandic Center for Research - Rannis, (2011) "Quality Enhancement Handbook for Icelandic Higher Education", retrieved on the 10th of January 2014 at: http://www.rannis.is/files/handbook_complete_1558767620.pdf

Lattuca, L. R., Terenzini, P. T., & Volkwein, J. F. (2006). Engineering Change: A Study of the Impact of EC2000: Executive Summary. ABET, Incorporated.

Tucker, A. B. (1991). Computing curricula 1991. Communications of the ACM, 34(6), 68-84

Breakfast with ICT Employers: What do they want to see in our graduates?

Margaret Hamilton
School of Computer Science and Information Technology
RMIT University
PO Box 2476 Melbourne
Victoria 3001 Australia
margaret.hamilton@rmit.edu.au

Angela Carbone
Office of the Vice-Provost (Learning and Teaching)
Monash University
PO Box 197 Caulfield East
Victoria 3145 Australia
angela.carbone@monash.edu

Christabel Gonsalvez
Faculty of Information Technology
Monash University
Wellington Road, Clayton
Victoria 3800 Australia
chris.gonsalvez@monash.edu

Margaret Jollands
School of Civil, Environment and Chemical Engineering
RMIT University
PO Box 2476 Melbourne
Victoria 3001 Australia
margaret.jollands@rmit.edu.au

Abstract

In an increasingly globalised and competitive economy, there is a need to ensure that graduates have the skills, knowledge and attitudes to be not only work ready for today but *work ready PLUS* for tomorrow. Data from the Graduate Destination Survey (2012) show that 75% of ICT students get a job once they complete their degree. Although employment outcomes are influenced by external market conditions, students, employers and other stakeholders expect universities to help students maximise their potential to find suitable work, that is, maximise their employability.

Employability is achieved by developing students' technical and generic skills. The development of technical skills is difficult in the computing sector where it has been argued that the ICT fundamentals have changed so much and continue to change rapidly. This project aims to understand what employability skills ICT employers expect to see in our graduates. Data for this study was collected from ICT employers, invited to participate in an industry breakfast to discuss the employability skills they are looking for when employing graduates.

A qualitative thematic analysis has been used to analyse the data, and the findings suggest that employers want ICT graduates to have effective teamwork and communication skills, with flexible and adaptive attitudes, without being arrogant. This study is part of a larger nationally funded project by the Australian Government Department of Education, aimed at developing employability skills in disciplines with low employment outcomes.

Keywords: ICT Graduate Employability; SFIA skills framework; Computer Science Education.

1 Introduction

In an increasingly globalised and competitive economy, there is a demand to ensure that graduates have the skills, knowledge and attitudes not only to be work ready for today but *work ready PLUS* for tomorrow (Fullan and Scott, 2014). Graduates need to be able to successfully navigate the messy realities of the workplace. From the perspective of students, employers, governments and other stakeholders, it is the responsibility of universities to best equip students to maximise their potential, to enable them to find suitable work and excel in the workplace - that is, to maximise their employability. It is usually a combination of technical and generic skills that makes students employable. Yorke (2006) defines employability as a set of achievements, which include skills, understandings and personal attributes. It is these achievements that make graduates more likely to gain employment, and then be successful in their chosen career. Employment outcome, however, refers to a measure of the number of graduates that actually secure full-time jobs, which in addition to a student's employability, is often influenced by external market conditions.

In the discipline of ICT, there is perceived gap between what employers would like to see in ICT graduates and what skills the graduates actually have. Data from the Graduate Destination Survey (2012) shows that approximately 75% of ICT students find a job once they complete their degree. There are many challenges faced by students who enter the IT sector, and one of the main challenges as suggested by John Craven from DB Results (Craven, 2014) at a presentation to the Australian Council of Deans of ICT (on 8th May 2014) is that the ICT fundamentals have changed. Technically, the field has moved from a strong focus on application software to a range of business platforms, from pure requirements gathering to outcomes and continuous improvement of systems, and the industry continues to struggle with suitable methodologies for successful systems development, with a strong move from the waterfall approach to agile methods (Wilton, 2011, Okay-Somerville & Scholarios, 2013, Curtis & McKenzie, 2001). Organisations are excited by the opportunities and challenges provided by big data and the continuous technological advances. It is recognised that the technical skills are dynamic, shaped by new technologies and market demands, but in addition to the changing technical landscape, generic skills are equally as important. Consequently, there is an expanding dialogue between universities, industry and governments around "graduate attributes" and "employability".

This paper reports on a component of a broader national study. The national study focuses on developing employability skills in our graduates across a variety of disciplines, with the specific aim of aligning the expectations of employers, professional bodies, academic staff, graduates and students. It seeks to identify good practice curricula that promote graduate employability. This large study is funded by the Australian Government Department of Education. The focus of this paper, however, is to understand more deeply contemporary employability skills and attributes ICT employers expect to see in our graduates. Section 2 outlines models for understanding employability. Section 3 outlines the data collection process from the ICT employers across Australia participating in the study. This sample reflects the diversity of ICT employers in Australia today, and the range of ICT disciplines. A detailed description of the employability skills that employers are looking for is provided in Section 4 of the paper, followed by a discussion of the findings in Section 5. The conclusion highlights the themes emerging from the data and future work from the study is foreshadowed.

2 Background

A review of the current literature generally differentiates between two general categories of employability skills: technical (job-specific, functional or discipline-specific skills) and generic (core or non-technical skills). (Bhaerman & Spill, 1988; Department of Industry, 2013; Lowden et al., 2011; Yorke, 2006).

While it is essential that individuals possess the technical skills necessary for their chosen profession in order to be considered employable in the industry, these skills seem to be "taken for granted" by employers (Yorke, 2006, p. 4). This is because it is generally assumed that graduates will have already acquired and developed these skills through their qualifications, hence possessing the technical skills simply becomes a "tick in the box" (Brown, Hesketh, & Williams, 2002, p. 19). This has resulted in the generic skills becoming a more important determinant of employability and the subject of much of the research literature.

2.1 Generic Skills

Generic skills or soft skills are understood to be the range of skills that encapsulate physical abilities, cognitive abilities and interpersonal skills that "enable people to succeed in a wide range of different tasks and jobs" (Yorke, 2006, p. 12). These skills are frequently cited as being an essential component of employability, highly valued by employers (Finch et al., 2013), and often encompass capabilities such as: written and oral communication, listening skills; professionalism, and teamwork and leadership. Alongside these skills, cognitive abilities such as problem solving, strategic and critical thinking and creativity are also considered to be essential components of employability (Lowden et al., 2011). In addition, Muhamad (2012) understands these skills as the ability to process complex information, question and reason and put new knowledge into practice. Furthermore, self-management, punctuality and time management, as well as the ability to adhere to workplace expectations are also important factors (UKCES, 2009). These generic skills are often termed "transferable skills" as they are applicable across a range of contexts and disciplines (Muhamad, 2012). In recent decades there has been an increasing demand for generic skills in the workplace.

2.2 Employability Skills

There is a significant body of work that focuses on employability being linked to skills gained, and there have been many attempts to define and categorise the skills that employers consider to be valuable in the workplace, across a broad range of professions. In an effort to holistically define and categorise the aforementioned generic skills, the Department of Industry (2013), in collaboration with several Australian government departments, developed the Core Skills for Work Developmental Framework. This framework groups these skills into three clusters:

- Navigating the world of work, including being able to manage career and work life and navigate rights and protocols at work;
- Interacting with others, encompassing communication, listening and interpersonal skills;
- Getting the work done, incorporating the ability to plan and organise, make decisions, identify and solve problems and create and innovate.

Another straightforward, practical model, the CareerEDGE model, developed by Pool & Sewell (2007) enables employability to be understood by students, parents and careers advisers and includes the following five key components of employability: career development and learning; work and life experience; degree subject knowledge; understanding and skills; generic skills and emotional intelligence.

More recently, Fullan and Scott (2014) define core learning outcomes as the six C's of deep learning, that will give graduates the *PLUS* factor which will allow them to manage the complex realities of the workplace. These core skills which involve academic and personal/interpersonal qualities and capabilities, include: *Character* such as grit, tenancies and perseverance; *global Citizenship* - considering global issues based on deep understanding of diverse values; *Collaboration* - working in teams with strong interpersonal skills; spoken, written and digital *Communication skills*; *Creativity* - having an entrepreneurial eye for economic and social opportunities and *Critical thinking* - being able to evaluate knowledge and apply it in the real world.

In the ICT profession, the Skills Framework for the Information Age (SFIA), a framework for describing and managing the skills needed by IT professionals, was developed by people experienced in the management of skills in IT. SFIA has become a de facto standard around the world, with over 2,500 corporate users in 195 countries (http://www.sfia-online.org/).

SFIA maps out 96 professional IT skills, organised in the following six categories - strategy and architecture; business change; solution development and implementation; service management; procurement and management support; and client interface. It also defines seven levels of attainment - follow; assist; apply; enable; ensure and advise; initiate and influence; and set strategy, inspire and mobilise, each of which is described in generic, non-technical terms. Each skill has an overall definition, and an “at-level” definition for each of the levels at which it can be recognised. “IT professional capability comes from a combination of professional skills, behavioural skills and knowledge. Experience and qualifications validate and support that basic capability” SFIA (2014).

SFIA has been adopted by the Australian Computer Society (ACS), as well as other professional societies and organisations. It provides a foundation for the professional grades, accreditation and programs of the ACS, as well as a common framework which allows an international understanding of what an ICT role actually involves. One aspect of the ACS accreditation process aimed at encouraging the development of generic skills, is the mandatory requirement that every undergraduate and postgraduate program include a final year capstone project. This project should aim to draw together all the technical skills the graduate has learned throughout their degree, together with a strong focus on the development of generic skills.

2.3 The Gap

It has been suggested that the employability skills acquired at university may not match the skills needed in employment (Wilton, 2011, Riebe & Jackson, 2013). Many employers are not satisfied with the skills graduates bring to the workplace. Research undertaken for the Council for Industry and Higher Education by Archer and Davison, (2008); explains that almost a third of employers have problems with graduates’ generic employability skills such as working in a team, communication, problem solving and self-management. A quarter of them are also disappointed with graduates’ attitude to work, while close to half of the employer are looking for business awareness and foreign language skills. Their report highlights the findings from a pilot survey of 233 employers and shows that there is a need for action by universities, employers, students and government to address both the reality and perception of the skills deficit in our graduates (Archer & Davison, 2008). This is a reality felt by both students and employers, and should be the impetus for policy makers and the Higher Education sector to address this gap.

Finch et al (2013), recognise that there needs to be a stronger relationship between education and employability, driven by an understanding of the factors that influence an undergraduate student’s “successful transition into the labour market” (p.682). Jollands et al (2012) also argue that employment outcomes can be enhanced by educational approaches which integrate generic skills related to employability into the curriculum. This study aims to develop a closer alignment between what employability skills ICT employers want, and what employability skills ICT academics need to develop in their students. The first step is to develop a contemporary understanding of the skills employers are looking for.

3 Method

3.1 Data Collection

Employers and staff of professional bodies were invited to participate in an Industry Breakfast Forum in June 2014. These were drawn from local employers from the project team member networks, program advisory committees from our respective Schools, and professional bodies for each discipline. Invitations were sent to staff in enterprises with a range of sizes, including small, medium and large companies.

ABCD University drew on employers from a wide range organisations that participate in their Industry Based Learning (IBL) program, which includes six month industry placements as part of students’ undergraduate degree programs. A vital part of the program is regular engagement with the IBL industry partners. Structured engagement is facilitated through 4 steering committee meetings held to help manage the IBL program and discuss IT graduate employment issues. The initial call to partners to attend the event was made at an IBL steering committee meeting, where partners were informed of the national project. A similar process was followed for employers of XYZ ICT graduates.

The Forum commenced with a breakfast, in a large meeting room, where employers from five disciplines were introduced to each other and members of the research team for each discipline. There were researchers and employers from Engineering, Psychology, Media Communications, Applied Science as well as from the ICT industry. The project leader addressed the whole group giving details of the project to all the employers. The agenda was semi-structured. The five teams then separated into smaller industry-based focus group sessions. Each focus group moved to a nearby separate room and the session was recorded.

The ICT focus group was facilitated by two project team members. Participants filled in a short demographic questionnaire, and signed their consent form. They were then asked to introduce themselves and then asked to consider the following three questions regarding graduate employment. The three key questions discussed were:

1 What are the key skills you are looking for in prospective employees?

2 What are the key attitudes you would like graduates to display?

3 What would you not like to see in the prospective graduates?

Employers were first asked to jot down on Post-it notes the key skills employers looked for in graduates. They then discussed how they might assess these skills in graduates during selection interviews. Employers provided short written responses on Post-it notes, which were then sorted on butchers paper on the wall.

In a second round of discussion, each employer jotted on Post-it notes, their ideas about what attitudes they looked for in graduates. They then discussed how they might identify these attitudes in graduates, and which ones were often lacking.

Similarly the third question above was answered by employers jotting ideas on Post-it notes, which were once again sorted onto butchers paper on the wall.

3.2 Data Analysis

A total of 11 employers attended the industry breakfast. The industries represented in the focus group included primarily medium to large multinational and local organisations from the professional services, insurance, financial, technology and retail sectors.

The employers provided written responses on sticky notes to each question. These were clustered according to emerging themes and analysed quantitatively. A summary of the Post-it note analysis is provided in Tables 1 - 4 in the results section 4.1 of this paper. The discussions during the focus groups were recorded and were transcribed verbatim. The transcriptions were entered into NVivo and analysed thematically with a qualitative open coding approach based on themes drawn from the Dacre Pool and Sewell employability framework (2007).

4 Results and Discussion

4.1 Results Classification

Employers' comments were sorted and similar attitudes and skills were grouped. The top two skills and attitudes identified by employers on the Post-it notes (as shown in Table 1 below) were Communication skills followed by Teamwork. Nine of the eleven employers identified communication skills as the top "employability" skill. The top two attitudes were "motivated and driven" and "flexible and adaptable". Employers did not want to employ graduates who were unwilling to be flexible, which is understandable in an ever changing ICT environment.

Post-it Note Summary	**Most Common**	**#**	**Second**	**#**
Skill	Communication skills	9	Teamwork	5
Attitude	Motivated & driven	6	Flexible and adaptable	6

Table 1: The top two skills and attitudes ICT employers look for in prospective graduate employees

Classified lists of the skills and attitudes that employers want to see in graduates, and what employers do not want to see in graduates are presented in Tables 2 - 4. In each of these areas, Post-it notes were sorted by the research team based on similarity, and a classification scheme was created which is represented in the first column. The number of comments in each classification is specified in the second column, and the employers' comments (with duplicates removed) are provided in the third column.

Skill classification	**No. of comments**	**List of comments on post-it notes**
Communication skills	9	- Strong written skills - Presentation/oral skills - Communication (verbal and written)
Teamwork	5	- Team player - Interacting with others - Team work
Problem solving	4	- Structured problem solving ability - Critical thinking - Analysing and problem solving
Business acumen	4	- Must have business acumen - Commercial awareness - Link technology to business (impact)
Technical ability	4	- Demonstrate IT aptitude - Relevant technical abilities ie. R, C++ - Comp Sci / Programming skills
Leadership	3	- Demonstrate leadership skills - Influences others - Influencing skills
Work experience	3	- Any work experience, can be part-time, doesn't have to be relevant - Industry based learning is a clear advantage - Industry knowledge
Project management	2	- Time management - Prioritising
Relationships	2	- Client focused - Good networker
Company knowledge	1	- Research into company and specific role

Table 2: Skills ICT employers are looking for in graduates

Table 2 presents the top skills as identified by the employers from their Post-it notes, while Table 3 gives the key attitudes they are looking for in prospective graduates.

Attitude classification	No. of comments	List of unique comments on post-it notes
Flexibility	6	- Adaptability - Long term thinking - Reliance and adaptability - Demonstrate ability to adapt to difficult individuals/ circumstances
Motivation	6	- Personal drive - Self motivated - Passion and drive to succeed - "can do" attitude (not all about themselves and what they can get out for themselves)
Initiative	3	- Innovative thinking - Demonstrate taking initiative
Self awareness	3	- Confident (willing to contribute) - Personality - Self aware - Works well under pressure
Learning ability	3	- Willingness to learn - Willing to learn / develop
Work experience	2	- Any extra curricula / work - Experience outside of papers - It's not all about getting the top grades: looking for an individual with extra curricular activities

Table 3: Attitudes ICT employers are looking for in graduates

While discussing the skills and attitudes employers were seeking from graduate employees, they discussed some of the downsides of identifying these during interviews. They addressed the pros and cons of large and smaller sized group interviews. They were then asked to itemise the qualities they would not like do see graduates display in these interviews and these are presented in Table 4 below.

Attitude classification	No. of comments	List of unique comments on post-it notes
Inflexibility	5	- Unwilling to be flexible - Inflexible, unwilling to compromise
Unprofessionalism	2	- Lack of professionalism (business sense); includes tardiness, dress attire, inappropriate conversation - Poor workplace behaviour - not in line with values
Poor Communication	2	- Poor communication - eye contact, ask questions - Inability to listen/absorb
Under-preparation	2	- Lack of preparation - not doing their research prior to coming through process - Lack of career direction or organisational knowledge
Arrogance	2	- Arrogance, too cocky - Arrogance (expectation that everything will be handed on a plate)
Lack of Initiative	2	- Don't wait for instructions - be inquisitive - Lack of self awareness
Lack of Confidence	2	- Apprehensive, fear to speak up
Lack of Trust	1	- Ability to trust the individual - do they have their own agenda?

Table 4: What ICT employers do not want to see in prospective ICT graduates

4.2 Round Table Focus Group Discussion Analysis

The themes that employers deem important which emerged from analysis of the focus group discussion are teamwork, communication skills in particular listening, business acumen, flexibility and adaptability, confidence (not arrogance). We discuss these in more detail below.

4.2.1 Communication Skills

A key skill all employers agreed was essential for a future employee was ‘communication skills’, and in particular they singled out 'listening' when asked to identify the most important one. They would like employees to listen to what is being asked of them by the employer, and not to tell them what needs to be done. They want someone who will not say the wrong thing at meetings and client interviews, but listen actively and contribute appropriately. Some illustrative quotes from employers include:

- *We look for the written communication skills [P1,p1]*
- *I think being a part of the good communicators [P8,p8]*
- *and often people fail an interview stage because they don't listen to the question and answer it completely different question to what you've asked… [P9,p8]*
- *Showing that empathy having that ability to sit down, listen, and talk, is something you want to see valued more often than not. Call it exuberance or enthusiasm or you know call it I'm excited to start a new role and I've got a tell you what it's [P1,p9]*

4.2.2 Teamwork

Another key skill which most employers identified as important was teamwork. As a first step, many employers interview their prospective employees in large groups of between 12 to 20. They are looking for people who can still "hold their own" with so many other people trying to contribute. However, they also look for someone who will not take over the group, and arrogantly promote themselves, their work or what they can do, the whole time. They want good listeners.

In the smaller group interviews of approximately 3 to 8 people, they are looking for someone who can explain their contribution to a group project, such as the capstone project which all IT graduates must complete before the end of their degree. Here they are looking for someone who has reflected on their own contribution, recognising they have not done the whole project, and are able to articulate why their contribution as part of a team was crucial to the success of the project.

Hence they are not looking for arrogance, where someone might say they were the manager and the success was entirely because of them, but nor do they want someone who will sit through such an interview and not promote themselves at all. In a group interview, they are not wanting someone to say they have a plan to be a manager in 5 years. Some quotes from the round table discussion which support the valued notion of teamwork include:

Some quotes which support this are:

- *At Company A and other organisations … it's the ability to work with others and have some thinking ability. You can see … maybe strategic thinking… longer term thinking, but also being able to work with others in the team [P3,p1]*
- *At Company B, we're very much a consulting services business now, we still have our technical software engineers and that type of thing when we look for these technical skills but the majority of our business is consulting and services so it's about finding people that have the ability to work well with our clients [P4,p1]*

4.2.3 Flexibility and Willingness to Learn

Attributes that employers liked to see are flexibility and a willingness to take up whatever role needs to be undertaken at that time in the business. They want employees who are comfortable being thrown in the deep end, where they have to quickly adapt to new environments and learn new things. Some companies have six month rotations through various parts of their organisation, so that staff are given the opportunity to identify areas of interest, and employers are able to observe where the employee would make the best contribution. They are looking for someone who is enthusiastic and willing to contribute. Some employer comments which illustrate this are:

- *I think it all comes back to that willingness to learn… More often and what we look for is someone who has that willingness to move from one side to another. [P1,p4]*
- *We need them to be adaptable. We need them to be willing to learn …but being adaptable is something that is really important to us [P4,p5]*
- *They might not have that similar role in a years' time and so they need to be adaptable. They need to be able to go hey this is a positive thing, not oh I thought I was going into SAP and not going to SAP anymore. [P9,p6]*
- *So you need to have the attitude that they want to be able to know that that's part of their development. [P3,p5]*

4.2.4 Business Acumen

Employers expect graduates to have a sense of business awareness so they can hit the ground running and enhance the worth of the company.

- *It might be their very first time working in a corporate environment but to have some sort of business awareness and sort of business savvy before they hit the workforce in terms of different stakeholders and knowing where people fit in the company. [P9,p6-7]*
- *At Company D as well…. more often than not they work with the business, they need to get that business prac and they need to understand staff we're working with cause we're working [as though] with gold so their merchandising is important. [P1,p10]*

4.2.5 Confidence but not Arrogance

The main thing which employers do not want to see is arrogance, which potential employees sometimes demonstrated during the interview process, by dominating the discussion. They do not want students telling them what their business needs are, and all the things they would change when they joined the organisation. Nor do they want to hear about all the wonderful things they can do technically. They believe that most students have the technical skills required by their company by the time they come to the interview, as they have passed their degree, or they have passed an entry level test, or both.

- W*e have grads who come in with a little bit of an attitude of expectation of what they want and you know we want to be a manager within a year or two. [P4,p5]*
- *We didn't like anyone who was too arrogant but who didn't contribute so it was one of those fine lines, you wanted someone who can contribute but if they started taking over, then that was thinking oh maybe we're not interested. [P2,p1]*
- *Mine's more so about unrealistic expectations, unwilling into compromise. [P4,p7]*

Overall ICT employers are looking for someone who demonstrates that they have the teamwork skills to fit in with their existing employees, belong to the groups by listening, identifying what they can contribute and generally overall enhance and improve their company.

5 Conclusion and Future Work

Students, employers and other stakeholders expect universities to help students maximise their potential to find suitable work, that is, to maximise their employability. In order to do this, it is necessary to work in partnership with industry and professional bodies and to understand the changing market conditions for graduates in their discipline. Students can best improve their generic skills when they and their teachers fully understand ICT employers' needs and expectations.

SFIA has provided a good framework for defining the employability skills students need, but it has yet to be fully integrated into academic programs. Capstone projects, a requirement for credentialing by the ACS, combine the technical skills required by future employers and generic skills. However, employers believe the generic skills need further, more guided and directed development, with a stronger emphasis on the listening skills, but without neglecting the oral and written skills.

This study provides a first step into understanding what contemporary employers are looking for in the ICT graduates. Our findings support previous studies (Archer & Davison, 2008, Wilton, 2011, Department of Industry, 2013) that ICT employers are not focused on the technical aspects when selecting employees. They believe that all graduates do have the foundation technical skills, but these technical skills are only deemed important during the selection process for highly technical roles. Universally, they were far more concerned about assessing the generic skills, which they believed were vital for sustained, successful careers in their organisations. These findings support and highlight the need for ICT degrees to continue to provide strong technical foundations, but to ensure that students are given every opportunity to develop the generic skills to improve employability outcomes.

Employers tell us that the key skills required are problem solving, business acumen and project management. They rated teamwork and communication skills as the top two skills, and within communication skills, good listeners were highly sought after. With teamwork, the ability to reflect and identify contribution to a team was highly prized. In addition, the key attitudes of self-awareness, learning, flexibility, initiative, motivation were highly regarded by employers as markers of successful staff. Arrogance and an inability to speak up when necessary, or speaking out when inappropriate are some of the things which employers did not want to see in ICT graduates.

The next stage of the project will focus on consultation with academic staff and undergraduates to assess the impact of activities to develop employability skills. A series of focus groups will be undertaken with undergraduate students, to collect evidence about when and where students are developing generic skills. This will be followed by a series of workshops with staff responsible for curriculum design to map teaching activities and generic skill development in undergraduates. Following this series of interviews with academic staff will be undertaken to document good practice case studies. The study will conclude with interviews with graduates to seek more in depth views on generic skills required, perceived gaps and strategies to redress the gaps.

6 Acknowledgements

The project has been funded by the Australian Government Office for Learning and Teaching (OLT) Grant. We thank the project team, research assistants and ICT industry employers who have given their time to participate in this project.

7 References

Archer,W. & Davison, J. (2008). Graduate employability: what do employers think and want? London: The Council for Industry and Higher Education (CIHE), http://www.brunel.ac.uk/services/pcc/staff/employability/?a=92718, last accessed 1/09/2014.

Bhaerman, R., & Spill, R. (1988). A Dialogue on Employability Skills: How Can They Be Taught? Journal of Career Development (Springer Science & Business Media B.V.), 15(1), 41-52.

Brown, P., Hesketh, A., & Williams, S. (2002). Employability in a knowledge-driven economy. In P. Knight (Ed.), Notes from the 13th June 2002 'Skills plus' conference, Innovation in education for employability, held at Manchester Metropolitan University.

Craven, J. (2014). ICT Resource Demand - An Industry View Power presentation to the Australian Council of Deans in ICT, UTS, Sydney 8th May 2014 http://www.acdict.edu.au/ALTA.htm (Last accessed 23 Aug 2014).

Curtis, D. & McKenzie, P., (2001). Employability Skills for Australian Industry Literature Review and Framework Development: report to Business Council of Australia, Australian Chamber of Commerce and Industry, 2001, http://www.voced.edu.au/node/1166, (last accessed 1/09/2014).

Department of Industry, (2013). Core Skills for Work Developmental Framework, http://www.industry.gov. au/skills/AssistanceForTrainersAndPractitioners / Core SkillsForWorkFramework/Documents/CSWF-Framework.pdf, (last accessed 1/09/2014).

Finch, D. J., Hamilton, L. K., Baldwin, R., & Zehner, M. (2013). An exploratory study of factors affecting undergraduate employability. Education + Training, 55(7), 681-704. doi: 10.1108/et-07-2012-0077.

Fullen and Scott (2014) New Pedagogies for Deep Learning White Paper, Education PLUS, Collaborative Impact SPC, Seattle, Washington.

Graduate Destination Survey (2012) – Graduate Careers Australia (GCA) Gradstats 2012, http://www. Graduate careers.com.au/research/start/agsoverview/ctags/gdso/, (last accessed 1/09/2014).

Jollands, M., Clarke, B., Grando, D., Hamilton, M., Smith, J.V., Xenos, S., Carbone, A., (2012). Developing graduate employability through partnerships with industry and professional associations, OLT grant, http://www.olt.gov.au/ (Last accessed 23 Aug 2014).

Lowden, K., Hall, S., Elliot, D., & Lewin, J. (2011). Employers' perceptions of the employability skills of new graduates: Edge Foundation, http://www.edge.co. uk/media/63412/employability_skills_as_pdf_final_ online_version.pdf, (last accessed 1/09/2014).

Muhamad, S. (2012). Graduate Employability and Transferable Skills: A Review. Advances in Natural & Applied Sciences, 6(6), 882-885.

Okay-Somerville, B. & Scholarios, D., (2013). Shades of grey: Understanding job quality in emerging graduate occupations, Human Relations 2013 66: 555.

Pool, L. D., & Sewell, P. (2007). The key to employability: developing a practical model of graduate employability. Education + Training, 49(4), 277-289.

Riebe, L., & Jackson, D., (2014). The Use of Rubrics in Benchmarking and Assessing Employability Skills, Journal of Management Education 2014, Vol. 38(3) 319 –344.

SFIA (2014) Skill Framework for the Information Age http://www.sfia-online.org/ (Last accessed 23 Aug 2014)

UKCES. (2009). The employability challenge : case studies, http://www.learningobservatory.com/resource /the-employability-challenge-case-studies/(last accessed 1/09/2014).

Wilton, N., (2011). Do employability skills really matter in the UK graduate labour market? The case of business and management graduates, Work Employment Society, 25, 85-100.

Yorke, M. (2006). Employability in higher education: what it is - what it is not. Learning and Employability Series 1, ISBN: 1-905788-01-0 The Higher Education Academy, http://www.employability.ed.ac.uk/docume nts/Staff/HEA-Employability _in_ HE(Is, IsNot).pdf, (last accessed 1/09/2014).

Computational Thinking, the Notional Machine, Pre-service Teachers, and Research Opportunities

Matt Bower
School of Education
Macquarie University
Australia
matt.bower@mq.edu.au

Katrina Falkner
School of Computer Science
University of Adelaide
Australia
katrina.falkner@adelaide.edu.au

Abstract

There is general consensus regarding the urgent and pressing need to develop school students' computational thinking abilities, and to help school teachers develop computational thinking pedagogies. One possible reason that teachers (and students) may struggle with computational thinking processes is because they have poorly developed mental models of how computers work, i.e., they have inadequate “notional machines”. Based on a pilot survey of 44 pre-service teachers this paper explores (mis)conceptions of computational thinking, and proposes a research agenda for investigating the use of notional machine activities as a way of developing pre-service teacher computational thinking pedagogical capabilities.

Keywords: Computational thinking, notional machine, teacher education, K-12

1 Introduction

Recent changes in ICT curriculum have moved from a focus on the use of ICT, i.e. digital literacy, to the need for awareness of how to create and influence the creation of new technologies. Recognition has grown, that in addition to the need to increase awareness and interest in Computer Science (CS), the fundamental concepts and skills of CS are valuable for children to learn. This has provided a driver for CS curriculum to be introduced as early as the first year of schooling. Preparing students to engage in current technologies and participate as creators of future technologies requires more than is currently being provided. We need to ensure that our educational systems provide not only the fundamentals of digital literacy – familiarity with the tools and approaches to interact with technology – but also the computational thinking processes needed to understand the scientific practices that underpin technology.

In alignment with recent global trends, the Australian primary and secondary school system is undergoing a significant period of change, with the introduction of a National Curriculum from K-10, new learning areas, and the development of national assessment programs. This new curriculum, defined by the Australian Curriculum Assessment and Reporting Authority (ACARA), identifies that “rapid and continuing advances in ICT are changing the ways people share, use, develop and process information and technology, and young people need to be highly skilled in ICT. While schools already employ these technologies in learning, there is a need to increase their effectiveness significantly over the next decade” (ACARA, 2012). The ACARA documents include ICT awareness (i.e. digital literacy) as a key capability, embedded throughout the curriculum, and introduce a new learning area, Technologies, combining the “distinct but related” areas of Design and Technologies and Digital Technologies (DT) (ACARA, 2013a). DT explicitly addresses the development of computational thinking skills as core to the understanding of digital technologies.

The success with which the digital technologies curriculum is implemented will depend, to a large extent, on the quality of learning and teaching. Consultation with Industry, Community and Education within Australia (ACARA, 2013b) has identified significant concerns in relation to teacher development (particularly at K-7), appropriate pedagogy, and skills needed for integration of DT learning objectives with the teaching of other learning areas. Approximately 55% of respondents indicated concern with the manageability of the implementation of the proposed curriculum, while 45% of respondents did not think that the learning objectives were realistic. Support for the professional development of teachers, including the creation of community networks to share insights and pedagogical approaches and research, has been identified as crucial in expanding CS curricula (Gander, et al., 2013). Bell, Newton, Andreae, & Robins (2012) describe the New Zealand experience of the rapid introduction of a senior secondary CS curriculum, and the need for extensive teacher development that addresses both content knowledge and pedagogical knowledge. However, many of the teachers who will be responsible for teaching the DT curriculum have not completed any studies that encompassed computational thinking concepts or processes, let alone how to teach these.

A classic concept in the computing education literature relevant to computational thinking is that of the “Notional Machine” (du Boulay, O'Shea, & Monk, 1989). The Notional Machine is a mental model that enables its user to make predictions about how a machine will perform. Without an adequate notional machine it is not possible to perform computational thinking processes (as elaborated later in this paper). Based on a pilot study of 44 pre-service teachers, this paper analyses conceptions and misconceptions of computational thinking, and based on the survey results and literature review proposes a research agenda for developing computational thinking capabilities based on notional machine activities.

2 What is Computational Thinking?

Computational thinking, as defined by Wing (2006) is: "*solving problems, designing systems, and understanding human behaviour, by drawing on the concepts fundamental to computer science*". Computational thinking involves understanding the fundamental concepts and abstractions that underpin computer science, and then reformulating problems into a form that can be solved readily using what we already understand. ACARA defines computational thinking as "*a problem-solving method that involves various techniques and strategies, such as organising data logically, breaking down problems into components, and the design and use of algorithms, patterns and models*" (ACARA, 2012). Understanding computational thinking involves understanding core computer science concepts, and the ability to conceptualise and create abstractions that define solutions to problems. But why is it important that we understand computational thinking? Why do we need to develop these mental models as part of our education system?

In the US, a recent survey of CS education at High Schools identified that Schools are "failing to provide students with access to the key academic discipline of CS, despite the fact that it is intimately linked with current concerns regarding national competitiveness…" (Gal-Ezer & Stephenson, 2009). Furthermore, recent reports from the US and Europe have argued that it is essential that children be exposed to CS concepts and principles from the very start of their education so that "every child [may] have the opportunity to learn Computing at School" (Gander, et al., 2013; Wilson & Guzdial, 2010). If not, we face the risk of our youth being placed in the position of consumers of technology produced elsewhere, unable to actively participate as producers and leaders in this field (Gal-Ezer & Stephenson, 2009; Gander, et al., 2013; Wilson & Guzdial, 2010). As Alan Noble, Engineering Director for Australia and New Zealand notes, "there is a difference between using a smartphone and creating an app that reaches millions of people" (Noble, 2012).

New curricula introduced in England (British Department for Education, 2013), Australia (ACARA, 2012), New Zealand and the new ACM CS standards (Seehorn, et al., 2011) have identified the need to educate for both digital literacy and CS, and the need to promote both learning areas from the commencement of schooling to support youth in participating in an increasingly digital society. Students who are exposed early generally have deeper interactions with computers, focused on exploring computers and related concepts rather than just utilising the computer for set tasks (Schulte & Knobelsdorf, 2007). Early exposure increases interest in computing by increasing computing self-efficacy (Akbulut & Looney, 2009).

However, it is also stressed that students would benefit from education in CS as an independent scientific subject on par with learning areas such as Mathematics or English (Gander, et al., 2013). It is essential that our education systems evolve, requiring the clear articulation of CS as a distinct discipline, including the integration of CS as a fundamental learning area across the curriculum and the exploration of the societal and cultural impacts of technology. Computational Thinking should be seen as an enabling subject (such as literacy or numeracy) whereas computing should be seen a separate discipline equivalent to Mathematics or Physics (BCS, 2010).

Developing capacity for computational thinking goes beyond building individual understanding and capabilities, however, but helps address a significant concern over the shrinking pool of qualified ICT professionals available to meet the demands of a rapidly growing industry. In a recent report by PWC (2013) on strategies and challenges in accelerating Australian innovation, they identify that "*Even if all international students were to stay in Australia post graduation, the supply of computer science and engineering graduates would still fall short of the numbers needed to accelerate growth*", while the Bureau of Labor Statistics (Lockard and Wolf, 2010), identifies that within computer and mathematical occupations, there is a 22.0% increase in employment projected from 2010-2020 (14.3% for all occupations).

3 Notional Machine

For many decades before developing computational thinking capabilities emerged as an important social agenda, Computer Science education researchers have been searching for the reasons why students find computing difficult. A foundational theory in computer science education that explains why students struggle to master computing concepts and processes is that of the "notional machine". The notional machine is an abstract version of the computer, "an idealised, conceptual computer whose properties are implied by the constructs in the programming language employed" (du Boulay, et al., 1989, p. 431).

The notional machine has been used in numerous studies (refer to Robins, Rountree, & Rountree, 2003, p. 149) and provides a theoretical orientation for examining how people think about computing and the misconceptions that may arise. That the notional machine assists learning is not a hypothetical proposition. For instance Mayer (1989) showed that students supplied with a notional machine model were better at solving some kinds of problems than students without the model.

In order for students to progress towards expert behaviour as efficiently as possible it is important to have an understanding of the difficulties they experience. This allows educators to provide scaffolding that helps learners to surmount these difficulties and allows the students themselves to pre-empt impediments to their learning by being aware of their potential before they arise. Du Boulay (1989) describes five inextricably linked potential sources of difficulty when learning computer programming:

1. general orientation (what programs are for and what can be done with them)
2. the notional machine (a model of the computer as it relates to executing programs)
3. notation (the syntax and semantics of a particular programming language)
4. structures (schemas and plans)

5. pragmatics (the skills of planning, developing, testing, debugging and so on).

Du Boulay et al. (1989) note that much of the early difficulty in learning computing arises from the student's attempt to deal with these different kinds of difficulties all at once. 'Misapplication of analogy', 'interaction of parts' and 'overgeneralisation' errors result. In the early stages teachers can assist the learning process by trying to address these domains separately (as far as possible) so as to reduce interference between them.

Du Boulay et al. (1989) suggest that in order for novice programmers to overcome comprehension problems caused by the hidden, unmarked actions and side effects of visually unmarked processes the notional machine needs to be simple and supported with some kind of concrete tool which allows the model to be observed. They suggest that the visibility component of such models be supported through 'commentary' – a teacher delivered or automated expose of the state of the machine. On the other hand the simplicity component of the machine can be supported through:

1. *functional simplicity* (operations require minimal instructions to specify)
2. *logical simplicity* (problems posed to students are of contained scale)
3. *syntactic simplicity* (the rules for writing instructions are accessible and uniform).

Du Boulay et al. (1989) conclude that matching visibility and simplicity components of notional machines to different populations of novice learners leads to improved educational outcomes. One would also suspect that without notional machine cognitive models, students' computational thinking progress would be severely restricted in the long term, and that the more sophisticated a student's notional machine the more developed their problem solving abilities. Both of these conjectures represent potential areas for further research.

As mentioned, the Notional Machine is a discipline specific mental model, and the literature on mental models also sheds light on how learning and teaching computational thinking may be enhanced. Norman (1983) distinguishes between the target system (the system that the person is learning or using), the conceptual model of the target system (an accurate and appropriate representation of the target system), the user's mental model of the target system (which may or may not be accurate and suffice), the researcher's conceptualization of the learner's model (a model of a model). Often teachers attempt to provide students with a conceptual model of a system to support the formation of students' mental models. Effective representations are those that capture the essential elements of the system leaving out the rest, with the critical point being which aspects to include and which to omit (Norman, 1993). Successfully selecting and describing the poignant features of a system allows students to concentrate upon the critical aspects of the system without being distracted by irrelevancies. When acquired, such conceptual models enhance students' capacity to reason and think. However if critical features are omitted or represented in a way that students misunderstand, then students may not comprehend crucial aspects of the system and may subsequently form misguided conclusions (Norman, 1993).

Some sub-domains of computer science have lead to specialised mental models of how students learn computing being developed. For instance, without a viable mental model of recursion that correctly represents active flow (when control is passed forward to new instantiations) and passive flow (when control flows back from the terminated instantiations) students cannot reliably construct recursive algorithm traces (Gotschi, Sanders, & Galpin, 2003).

There are several advantages to such domain specific models. Firstly, they can inform educators' decisions about the required approach to learning – in the case of recursion a constructivist approach is required in order for students to create a viable mental model adequate to apply design concepts and solve problems. Secondly, domain specific models assist lecturers by providing accurate mental models, such as Kayney's 'copies' model of recursion, that have been demonstrated as successful at promoting understanding. Thirdly, such research explicitly exposes non-viable mental models that students may form (such as the looping, magic, and step models), allowing lecturers and pupils to pre-empt student errors (Gotschi, et al., 2003).

4 Developing Computational Thinking

There are a variety of broad recommendations about how to develop computational thinking generally, most of which emanate from the Computer Science education literature. Pedagogues recommend connecting Computational Thinking to young people's interests (Resnick, et al., 2009), for instance, through computer games (Carter, 2006; Lenox, Jesse, & Woratschek, 2012) or multimedia based learning tasks (Blank, et al., 2003). A games based approach to introducing programming in the middle years has been shown to help develop computational thinking concepts (events, alternation, iteration, parallelism, additional methods, parameters, local and global variables) at the same time as enhancing students enjoyment of learning computing (Repenning, Webb, & Ioannidou, 2010).

Providing students with a low floor (easy to learn), high ceiling (hard to master, many opportunities to learn), wide walls (flexible and adaptable to a wide range of applications) enables students of different ability levels to remain engaged (Resnick, et al., 2009). Stephenson et al. (2005) recommend designing course materials that incorporate meaningful learning through the use of problem-solving approaches, appealing experimental environments, and an explicit emphasis on design and a real-world focus. Supporting skills beyond programming has been shown to increase student satisfaction with computing and may broaden further participation (Repenning & Ioannidou, 2008).

Creating a conducive learning environment has also been proposed as a way to enhance computational thinking. For instance, Stephenson et al. (2005) recommend establishing a welcoming environment that models life-long learning. Barr & Stephenson (2011) suggest increased use of computational vocabulary by teachers and students where appropriate, acceptance of failed solution attempts by teachers and students, and tasks involving team work by students. Yet, there is little research to substantiate these claims.

5 The challenge of teaching Computational Thinking

One of the main problems faced by the domain is that many students perceive computing to be essentially the same as technology training, which can be seen as repetitive and teaching skills that the students already know such as how to use standard Office tools (BCS, 2010). It is also possible that teachers (including pre-service teachers) may not always have a firm understanding of what computational thinking involves (as will be explored later in this paper).

Studies have identified increased anxiety and concern over preparation time when dealing with unfamiliar content (Curzon, McOwan, Cutts, & Bell, 2009). Even in cases where teachers are experienced with computing fundamentals, the integration of new tools can create anxiety that causes them to deviate from their planned lessons (Meerbaum-Salant, Armoni, & Ben-Ari, 2013).

Training teachers to teach computational thinking is an essential piece of the puzzle (BCS, 2010; Black, et al., 2013). Poor lessons demotivate learners, creating negative attitudes towards the subjects, and this can create a vicious cycle of demotivating teachers who in turn create poorer lessons (BCS, 2010). Professional development is critical in order for teachers to effectively develop computational thinking pedagogies, (Barr & Stephenson, 2011). This is not only about offering training courses, but also establishing effective communities of practice to provide ongoing support and sharing of resources (Black, et al., 2013).

It is also critical to provide resources to help teachers effectively teach computational thinking concepts and processes (Barr & Stephenson, 2011). Settle et al (2012) identify specific difficulties for educators in translating materials into existing curriculum, with an emphasis on the increased difficulty in adopting and integrating new tools. It is challenging is to provide teachers with material which effectively conveys the most important aspects of computing without reducing it to tool use or programming, both of which are misconceptions of computing (Battig, 2008). Tinapple, Sadauskas, & Olson (2013) further comment on the challenge of implementing lessons where expected software and/or hardware are not easily available.

Another issue is that teachers often utilise fun activities with a focus on impressive technology, physical computing and programming using constructionist environments rather than providing opportunities for deep learning of computational thinking (Black, et al., 2013). These results are indicative on a focus on tool usage for engagement, rather than a deep understanding of computational thinking processes and concepts.

6 A pilot survey of pre-service teachers

In order to gauge pre-service teachers' perceptions of computational thinking learning and teaching in light of the upcoming Australian Digital Technologies Curriculum a pilot survey was run in April of 2014. The anonymous online survey was issued to 84 pre-service teachers who were completing the 300 level subject "EDUC362 – Digital Creativity and Learning" at Macquarie University. The survey was conducted during Week 3 of Semester 1 (March 2014). A total of 44 pre-service teachers volunteered to respond. Apart from demographic questions relating to age, gender and the program of study in which the student was enrolled, the survey asked about pre-service teachers' awareness of the upcoming Australian Digital Technologies curriculum, their conceptions of the term 'computational thinking', their understanding of pedagogies and technologies that can be used to develop computational thinking, and their confidence to teach computational thinking.

Open-ended responses were analysed using qualitative coding techniques. First classified using an open-coding phase to determine preliminary analytic categories. Next, axial coding was carried out to determine emergent themes and refine categorisations. Lastly, a selective-coding phase supported representation of the conceptual coding categories for reporting purposes. (See Neuman, 2006, for further details of the approach.) If responses addressed multiple issues they were coded in more than one category, meaning that it was possible to have a greater tally of responses across the items than the number of respondents.

Quantitative data was interpreted and reported using standard descriptive statistics techniques.

6.1 Results

Of the 44 students who chose to respond, 38 were intending to be primary school teachers and 5 were planning to be secondary school teachers (2 science, 2 languages, and 1 english/history). On respondent did not indicate their intended teaching level. The large majority of respondents were in their third or fourth year of their program (42 out of 44). The age distribution was right skewed with 29 participants indicating that they were in the 18-24 age range. A total of 33 females and 11 males participated.

6.1.1 Awareness of Computational Thinking

Pre-service teachers' awareness of the upcoming Australian Digital Technologies Curriculum (ADTC) and whether they had heard of the term 'computational thinking' is shown in Table 1.

	Heard of 'Computational Thinking'	Not heard of 'Computational Thinking'
Aware of ADTC	15	11
Unaware of ADTC	11	7

Table 1: Awareness of the upcoming Australian Digital Technologies Curriculum (ADTC) and the term 'computational thinking'

The table demonstrates that an awareness of the upcoming ADTC did not necessarily imply an awareness of 'computational thinking', even though computational thinking was highlighted by the Australian Curriculum Assessment and Reporting Authority (ACARA) as a distinguishing core feature of the ADTC. Similarly, awareness of computational thinking did not necessarily derive from the ADTC, with a quarter of students indicating that they had heard of computational thinking but did not know about the impending ADTC.

6.1.2 Conceptions of 'computational thinking'

There were 32 pre-service teachers who chose to respond to the question "what does computational thinking mean to you?". Table 2 summarises their responses into the categories that emerged from the coding process. Note once again that some responses are tallied under two or more categories if the response incorporated multiple elements. 'Problem solving using technology' has been included as a separate category to 'problem solving' or 'using technology' as it demonstrates a deeper understanding of computational thinking than either of the latter two categories.

Computational thinking construct	fn
problem solving using technology	11
using technology	10
technological thinking	5
logical thinking	5
gathering/organising/processing information	3
analytical thinking	3
critical thinking	2
creative thinking	2
mathematical thinking	2
problem solving	2
thinking like computer	2
scientific thinking	1
structured thinking	1
strategic thinking	1
testing	1
efficiency	1
non-descript	1

Table 2: Summary of pre-service teacher conceptions of 'computational thinking'

Over one third of respondents described computational thinking as involving "problem solving using technology", though descriptions varied widely in sophistication. For example:

Problem solving using technology; using technology in a variety of ways to approach a problem; analysing and logically organising data, generating problems that require computers assistance; identifying, testing, and implementing possible solutions

Using computer technology to solve a problem.

Having heard of the term computational thinking did not necessarily result in more sophisticated responses being provided. For instance, the first response above is from someone who had not heard of computational thinking and the second response is from someone who had.

Nearly one third of the pre-service teachers considered computational thinking to merely be using technology, for instance "awareness of how to operate software, ability to 'self help'". Two students described it as problem solving without associating it with technology, and one student had a blurred conception of computational thinking as both digital literacies and problem solving using technology: "Digital Literacy, the ability to use technology to solve problems and assist learning, create digital artefacts".

Pre-service teachers were able to identify types of thinking associated with computational thinking, namely logical thinking, analytical thinking, critical thinking, creative thinking, mathematical thinking, scientific thinking, structured thinking, and strategic thinking. Some were able to identify activities and concepts associated with computational thinking, such as testing, efficiency, gathering information and organising data. Only two students were able to associate computational thinking with more than three of any of the above elements.

Two pre-service teachers erroneously thought computational thinking was thinking like a computer, for instance "Thinking or memorising in a way that computer works". One pre-service teacher gave the non-descript response "a process or a way of thinking to understand topics". There were five students who used the term "technological thinking" or synonymic phrases, which has no clear meaning,

6.1.3 Associated Pedagogies

There were 30 pre-service teachers who chose to respond to the question "What pedagogical strategies do you have (or can you think of) for developing school students' computational thinking capabilities?" Their responses are summarised in Table 3.

pedagogical strategies	fn
using technology	13
group work	6
problem based tasks	6
active learning	4
direct instruction / modelling	3
inquiry based approach	3
games/play	2
none / non-descript	2
provide scaffolding	2
teacher familiarity with technology	2
authentic problems	1
brainstorming	1
establish purpose	1
provide process for thinking	1
safe environment	1
writing code	1

Table 3: Summary of pre-service teacher pedagogical strategies to develop computational thinking

The most popular pedagogical strategy represented in students' responses (n=13) was to simply use technology, for instance: "Continuous practice, engagement and exposure to different computer technology". Four of these responses also mentioned problem solving in association with the use of technology. Six students made general mention of how group work strategies could be used (for instance, collaborative learning, cooperative learning, paired learning). There were sixteen instances where responses discussed the nature of the learning process (problem-based learning, active learning, inquiry learning, games based learning, brainstorming, writing code). It is interesting to note that only one pre-service teacher mentioned writing code. There were another ten cases where responses discussed the role and responsibilities of the teacher (direct instruction / modelling, provide scaffolding, be familiar with technology, establish purpose, provide processes for thinking, creating a safe environment).

Overall responses were lacking in detail so in most cases it was difficult to tell whether pre-service teachers had a concrete understanding of how the pedagogy could be applied to develop computational thinking. Responses also revealed more about pre-service teacher conceptions of computational thinking. For instance, one respondent's pedagogical strategies were:

Using group work (heterogeneous groups) for students to engage in negotiation, reasoning and student discussion. I would also use apps for students to engage in thinking abstractly and outside of the square such as Comic life, I-movie.

It is unclear how this respondent would use group work to develop computational thinking, and it appears that while the student did associate abstraction with computational thinking, they did not appear have a clear understanding of how technology could be used to develop computational thinking.

6.1.4 Supportive Technologies

Asking pre-service teachers the question "*How can technologies be used to help develop school students' computational thinking capabilities? (Provide specific examples if you can.)*" offered further insight into their conceptions of computational thinking (see Table 4). Of the 26 students who responded to this question, 10 provided only unspecific suggestions about how technology could be used to develop computational thinking, for instance "organise and help the logical thinking". Six students talked generally about how technology could be used to increase engagement, for of which were from the unspecific respondents. For example: "technological resources can be more engaging/exciting to students".

Technologies to support computational thinking	fn
unspecific	10
engagement	6
conduct research (e.g. searching Internet)	5
presentation tools	4
software/apps - general	4
comic/story creation tools	3
mindmapping	3
create 3D objects	2
data analysis (e.g. spreadsheet)	2
practice - general	2
brainstorming software	1
none	1
program creation	1
publishing tools	1
websites	1

Table 4: Summary of pre-service teacher identified technologies for developing computational thinking

Some pre-service teachers provided more specific suggestions about how technology could be used to develop computational thinking, but for many of these it was unclear how it actually would develop computational thinking. For instance, using comic/story creation tools, mindmapping tools, brainstorming and presentation tools are not obviously and usually related to developing compuational thinking. The specific examples of technologies that pre-service teachers identified were Mindmeister, Comic Life, Toontastic. Prezi, iBooks, and Google Sketchup. Five students mentioned using the Internet for research purposes, and only one identified a technology that was specifically related to computational thinking (the code.org website).

6.1.5 Pre-service teacher confidence

There were 32 pre-service teachers who chose to respond to the questions relating to how confident they felt to develop their students' computational thinking capabilities (see Figure 1). From the graph it can be seen that 18 of the 32 pre-service teachers (56%) indicated that they were to some degree unconfident rather than confident about teaching computational thinking.

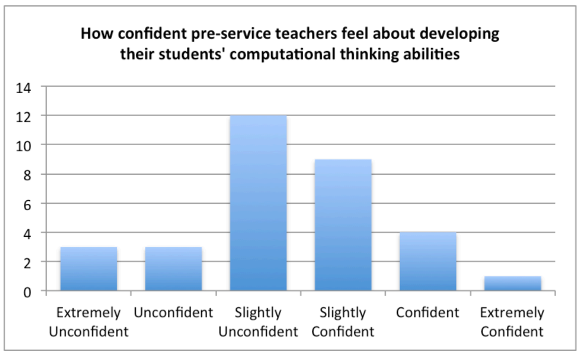


Figure 1: Pre-service teachers' confidence about developing their students' computational thinking abilities

It is important to note that responses on the confident side of the scale did not mean that pre-service teacher confidence was warranted. For instance, some pre-service teachers indicated that they were 'slightly confident' about developing their students'' computational thinking abilities, but had not heard of the term computational thinking and had poor conceptions of computational thinking such as:

Computational thinking is ones ability to navigate and problem solve using the medium of technology such as ipad, macbooks and IWB's.

teaching and learning using technology

More concerning, there were some teachers who had heard of computational thinking and indicated that they were 'confident' about developing their students' computational thinking abilities yet had erroneous conceptions of computational thinking, for instance:

using the computer to help with forming ideas and opinions / - how technology can help your thinking

6.1.6 Lack of Confidence

When pre-service teachers were asked "what prevents you from feeling confident about developing your students' computational thinking capabilities?" responses related to pedagogical issues, technology issues, general issues, circumstantial and affective issues.

Nine pre-service teachers felt unconfident about developing their students' computational thinking because of pedagogical issues, including unfamiliarity with the curriculum (5), lack of pedagogical strategies (3), lack of lesson ideas (1), and uncertainty how to apply computational thinking to real world situations (1). There were eight pre-service teachers who felt that they did not have the technological knowledge and experience to feel confident about teaching computational thinking, though many of these appeared to be confusing computational thinking with general technology usage (for instance "I lack ICT knowledge"). One of these pre-service teachers felt they did not have the required computer science and programming knowledge.

There were thirteen pre-service teachers who indicated more general reasons for their lack of confidence including a poor understanding of what computational thinking means (4), a general lack of knowledge (6) and a general lack of experience (3). Two pre-service teachers did not feel confident about teaching computational thinking due to circumstantial factors relating to becoming a teacher:

Still learning about being a teacher so not yet confident in any particular area

I wasn't not taught like this at school, content and the use of technology

One pre-service teacher spoke directly about the fear of the unfamiliar affecting their confidence:

Because it is something new to me and to teach something i am just coming to terms with slightly scares me and i lose confidence because of that

6.1.7 Building confidence

When pre-service teachers were asked "*What could help you to feel more confident about developing your students' computational thinking capabilities?*" the most common response related to explicit professional development (11 respondents). Other items identified by students provide insight into the form that such professional development might take. There were 6 pre-service teachers who indicated they would like a better understanding of pedagogical strategies, 7 who wanted greater exposure to and experience with technology, and 7 who felt that a better understanding of computational thinking would improve their confidence to teach computational thinking. There were seven students who indicated that greater understanding and practice generally would be beneficial.

Pre-service teachers identified other factors that could improve their confidence in developing computational thinking including more resources and information, learning more about computer programming, learning more about the research relating to computational thinking, and having well planned lessons.

6.2 Limitations of this study

A limitation of this study is that it was only issued to a small sample of pre-service teachers from one university, and results may vary widely depending on the institution. As well, students were not asked about their previous studies of computing, which one would expect would have a large influence on their responses. Any future iterations of the survey will ask students about their previous exposure to computing.

The survey was conducted before pre-service teachers completed a topic on computational thinking in the third year unit they were studying. This was done so that responses were more representative of the general pre-service teacher population of the university, most of whom do not complete the unit which was offered for the first time in 2014. After completing the unit student responses may have been quite different. However, it is conjectured that many universities do not yet have any courses that cover computational thinking as an explicit topic, and as such the responses may be more representative of the broader pre-service teacher population both nationally and internationally.

As this was an online survey students may not have been motivated to provide elaborate responses that accurately represented the full extent of their perceptions and conceptions. Semi-structured interview techniques may be necessary to probe more deeply into pre-service teacher thoughts surrounding computational thinking.

7 Discussion of results

Generally speaking pre-service teachers had a weak understanding of computational thinking. There are a large proportion of pre-service teachers who confuse computational thinking with using technology generally (for instance word processing or searching the internet). Pre-service teachers correctly associated computational thinking with problem solving using technology, logical thinking, gathering/organising/processing information, analytical thinking, critical thinking, creative thinking, mathematical thinking, scientific thinking, structured thinking, strategic thinking, testing and efficiency, though only two students were able to associate it with more than three of these points. This indicates that there is extensive potential to improve pre-service teachers' conceptions of computational thinking. The data also implied we should not assume that because pre-service teachers are aware of the upcoming Digital Technologies Curriculum they understand computational thinking, or visa versa – half of respondents were aware of one but not the other.

For many of the pre-service teachers the extent of their pedagogical strategies for developing computational thinking was simply to have students use technology. Collectively pre-service teachers were able to identify generally appropriate pedagogical strategies such as types of group work and student centred learning. Several teachers identified the role of the teacher in providing instruction and creating a conducive learning environment. Yet because responses were almost invariably lacking in detail there was no evidence to indicate that the pre-service teachers had specific and clear ideas about how to develop their students' computational thinking capabilities.

The technologies they identified to support the learning of computational thinking provided further verification that many students did not understand what was meant by computational thinking - the specific tools that were suggested (such as Comic Life and iBooks) bore no specific relation to computational thinking and only one student mentioned a purpose built platform (the code.org website).

Pre-service teachers were of varying confidence about teaching computational thinking, and some were overconfident based on their evidenced understanding. Not only were the majority on the unconfident side of the response spectrum, but several of those who indicated confidence had poorly formed or incorrect conceptions of what computational thinking actually meant. There were classic examples of third order ignorance (Waite et al 2003) where pre-service teachers were unaware that they did not know.

Responses from pre-service teachers indicated that they would value professional learning opportunities that focused on:

- Developing their computational thinking pedagogical capabilities - understanding of the curriculum, lesson ideas, strategies for implementation, links to real world examples
- Technological understanding - exposure to and practice with the sorts of technologies that can be used to develop computational thinking, and even elementary programming instruction
- Content knowledge - a better understanding of what computational thinking is and means.

This accords with the well renowned Technology Pedagogy and Content Knowledge (TPACK) model of teacher learning and practice (Mishra & Koehler, 2006). Responses also highlighted the need for both knowledge and practice. Some pre-service teacher responses highlighted the importance of affective considerations when designing professional learning - this is unfamiliar territory for many teachers who have never been taught or learnt computational thinking so it is important to sensitively scaffold their confidence.

8 Computational thinking research agenda

The notional machine has been an important and useful construct in computer science education (Robins et al, 2003) but there has been little if any work investigating how it can be used to understand and enhance computational thinking learning and teaching. There is urgent and pressing need to develop school students' computational thinking capabilities and teachers' computational thinking pedagogies (as established through the literature review and also by the data collected in this study). Thus there are several research opportunities to investigate how notional machines can inform our understanding of computational thinking and improve how it is learnt. Phrased as research questions, these are:

1. *How do notional machine constructs map to different computational thinking environments?* For instance, how do we define notional machines for computational thinking systems that may vary from Eclipse, to Scratch, to Beebots?
2. *How can 'visability' (du Boulay et al., 1989) be used to support computational thinking within computational thinking environments?* There may be several pedagogical strategies along the lines of including visual debugging-style output within programs to make the operations of the machine visible to students, thus enhancing their notional machine, but their effectiveness has not been investigated specifically from a computational thinking frame of reference.
3. *How can 'functional simplicity'(du Boulay et al., 1989) be best instantiated through easy to understand instructional sets?* This relates to the quality of introduction and explanation of how the machine works, and success may reside in illuminating exemplars, economical explanation, and powerful analogies). As Norman (1993) points out in order to help students form accurate mental models it is just as critical to decide what should be left out as what should be included.
4. *How can 'syntactic simplicity' (du Boulay et al., 1989) be fostered through accessible and uniform programming grammars?* This has been applied in some of the computational thinking tasks available through Code.Org, Scratch, Alice, and the like that use visual interfaces to write programs. Ideally teachers would utilise and even create non-computer based tasks that develop computational thinking abilities, in which case an understanding of syntactic simplicity is critical.
5. *How do we incrementally graduate the 'logical simplicity'(du Boulay et al., 1989) of the problems to be solved in line with the developing conceptions of the novice computational thinker?* (Scope and sequencing and timing issues are crucial so that students are neither bored nor overwhelmed - low floor, high ceilings, wide walls. Bower's Taxonomy of Task Types provides a one possible hierarchy for incrementing task complexity). The idea is to attempt to avoid problems relating to trying to learn about what computational thinking means, developing notional machines, learning languages, learning computing structures, and developing computational thinking process skills all at once (the 5 sources of difficulty identified by DuBoulay). Teachers need to know how to deconstruct computational thinking to avoid possible student cognitive overload.
6. *Where do 'misapplication of analogy', 'overgeneralisation' and 'interaction of parts' and potentially other types of errors commonly occur in the curriculum?* An understanding of these errors and where they occur helps teachers to better support the learning of computational thinking constructs. More importantly, how can we use these instances to create threshold learning experiences.

7. *How do researchers and educators accurately gauge novice mental models of target systems so that we can understand how to effectively guide learners towards correct conceptual models?* As Norman (1983) distinguishes between the correct conceptual model of the target system, the user's mental model of the target system, and the researcher's conceptualisation of the learner's model, understanding how to gauge and contrast these may be the key to understanding computational thinking learning and teaching, As Gotschi, Sanders and Galpin (2003) point out, domain specific models not only provide a point of reference to help identify non-viable mental models but also provide teachers with a resource to help develop their students' mental models.
8. *How do we best structure teacher professional learning in order to most effectively develop their computational thinking pedagogical capabilities?* This not only relates to the execution of professional learning courses, but also the development of an appropriate learning community around computational thinking pedagogy comprised of pre-service teachers, in-service teachers, researchers and developers. The pre-service teachers provide some general ideas, as does the literature, yet the devil will be in the detail.

Universities should be playing a key role in the development of teachers, methods and curriculum (Tucker, et al., 2003). A key element for a successful curriculum in schools is founding the resources and teaching practices on research into computer science education (Hazzan, Gal-Ezer, & Blum, 2008). In order to develop high quality computing curriculum is to have the course part of the research process, whereby teaching and learning data is used to iteratively refine the educational process (Hazzan, et al., 2008). Teachers can then become active participants in the research process. In Israel the teacher preparation process includes some research components, so that teachers can learn how to iteratively refine their teaching practices. In this way, research projects can contribute to the education of students, teachers and the educational community at large.

9 Concluding remarks

Accurate notional machines underpin successful performance in computational thinking. A structured rather than haphazard approach to examining notional machine understanding is required if we are to help students (and teachers) identify their misconceptions and take appropriate remedial action. Notional machine understanding is a prerequisite for effective teaching of computing, but not a guarantee. Teachers also need to have an appropriate repertoire of computational thinking pedagogies and technological knowledge in order to successfully teach computational thinking concepts and create a conducive learning environment for students.

This paper calls for further research into how the notional machine can be used to better understand and develop the computational thinking abilities of students as well as the computational thinking pedagogical capabilities of teachers. Results from this study suggest that pre-service teachers are ill prepared to teaching computational thinking, and need pedagogical strategies, experience with relevant technologies, and a better understanding of what computational thinking means. The computer science and education fields more generally need a greater understanding of how computational thinking is effectively learnt and taught in order to better support students and teachers.

The literature has identified visibility, functional simplicity, syntactic simplicity, logical simplicity and graduation as critical pedagogical issues, but how these relate to specific aspects of computational thinking learning is an open question. As yet there is no clear understanding of how to best describe and gauge notional machines, nor key places where novice misconceptions appear in the computational thinking curriculum. This paper is a call to action and an invitation to researchers interested in working on understanding the computational thinking research questions identified in this paper.

10 References

ACARA (2012). The shape of the Australian curriculum: technologies. Retrieved 17 August, 2014, from http://www.acara.edu.au/curriculum_1/learning_areas/technologies.html

ACARA (2013a). The Australian curriculum: Technologies information sheet. Retrieved 17 August, 2014, from http://www.acara.edu.au/curriculum_1/learning_areas/technologies.html

ACARA (2013b). Draft Australian Curriculum: Technologies Foundation to Year 10 Consultation Report. Retrieved 17 August, 2014, from http://www.acara.edu.au/curriculum_1/learning_areas/technologies.html

Akbulut, A. Y., & Looney, C. A. (2009). Improving IS student enrollments: Understanding the effects of IT sophistication in introductory IS courses. *Journal of Information Technology Education, 8*, 87-100.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *ACM Inroads, 2*(1), 48-54.

Battig, M. (2008). *Piltdown man or inconvenient truth? A two-year study of student perceptions about computing.* In Proceedings of ISECON.

BCS, T. C. I. f. I. (2010). *Consultation response to Royal Society's Call for Evidence – Computing in Schools*. The Royal Society: T. C. I. f. I. BCS.

Bell, T., Newton, H., Andreae, P., & Robins, A. (2012). *The introduction of computer science to NZ high schools: an analysis of student work.* In Proceedings of the 7th Workshop in Primary and Secondary Computing Education, (pp. 5-15): ACM.

Black, J., Brodie, J., Curzon, P., Myketiak, C., McOwan, P. W., & Meagher, L. R. (2013). *Making computing interesting to school students: teachers' perspectives.* In Proceedings of the 18th ACM conference on Innovation and technology in computer science education, (pp. 255-260): ACM.

Blank, G. D., Pottenger, W. M., Sahasrabudhe, S., Li, S., Wei, F., & Odi, H. (2003). Multimedia for computer science: from CS0 to grades 7-12. *EdMedia, Honolulu, HI.*

British Department for Education (2013). *The national curriculum in England*. Cheshire, UK: Crown.

Carter, L. (2006). *Why students with an apparent aptitude for computer science don't choose to major in computer science.* In ACM SIGCSE Bulletin, (pp. 27-31): ACM.

Curzon, P., McOwan, P. W., Cutts, Q. I., & Bell, T. (2009). *Enthusing & inspiring with reusable kinaesthetic activities.* In ACM SIGCSE Bulletin, (pp. 94-98): ACM.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: presenting computing concepts to novices. In E. Soloway & J. C. Spoher (Eds.), *Studying the Novice Programmer* (pp. 431-446). Hillsdale, NJ: Lawrence Erlbaum.

Gal-Ezer, J., & Stephenson, C. (2009). The current state of computer science in US high schools: A report from two national surveys. *Journal for Computing Teachers*, 1-5.

Gander, W., Petit, A., Berry, G. r., Demo, B., Vahrenhold, J., McGettrick, A., et al. (2013). *Informatics education: Europe cannot afford to miss the boat.*

Gotschi, T., Sanders, I., & Galpin, V. (2003). Mental models of recursion *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (pp. 346-350): ACM Press.

Hazzan, O., Gal-Ezer, J., & Blum, L. (2008). *A model for high school computer science education: the four key elements that make it!* In ACM SIGCSE Bulletin, (pp. 281-285): ACM.

Lenox, T., Jesse, G., & Woratschek, C. R. (2012). Factors influencing students decisions to major in a computer-related discipline. *Information Systems Education Journal, 10*(6), 63.

Lockard, C.B and Wolf, M. (2012). Occupational employment projections to 2020. *Monthly Labor Review* , January 2012, 84-108.

Mayer, R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway & J. C. Spoher (Eds.), *Studying the Novice Programmer* (pp. 129-159). Hillsdale, NJ: Lawrence Erlbaum.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education, 23*(3), 239-264.

Neuman, W. L. (2006). *Social research methods – Qualitative and quantitative approaches (6th Edition).* Boston: Pearson Education.

Noble, A. (2012). Science the key to seize control of the future (26th December). *Sydney Morning Herald.* Retrieved from http://www.smh.com.au/opinion/politics/science-the-key-to-seize-control-of-the-future-20121225-2bv55.html

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental Models*. Hillsdale, NJ: Erlbaum.

Norman, D. A. (1993). *Things That Make Us Smart*: Perseus Books.

PWC (2013). *The startup economy: How to support tech startups and accelerate Australian innovation.*

Repenning, A., & Ioannidou, A. (2008). Broadening participation through scalable game design. *ACM SIGCSE Bulletin, 40*(1), 305-309.

Repenning, A., Webb, D., & Ioannidou, A. (2010). *Scalable game design and the development of a checklist for getting computational thinking into public schools.* In Proceedings of the 41st ACM technical symposium on Computer science education, (pp. 265-269): ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., et al. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172.

Schulte, C., & Knobelsdorf, M. (2007). *Attitudes towards computer science-computing experiences as a starting point and barrier to computer science.* In Proceedings of the third international workshop on Computing education research, (pp. 27-38): ACM.

Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., et al. (2011). CSTA K-12 Computer Science Standards: Revised 2011.

Settle, A., Franke, B., Hansen, R., Spaltro, F., Jurisson, C., Rennert-May, C., et al. (2012). *Infusing computational thinking into the middle-and high-school curriculum.* In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, (pp. 22-27): ACM.

Stephenson, C., Gal-Ezer, J., Haberman, B., & Verno, A. (2005). The new educational imperative: Improving high school computer science education. *Computer Science Teachers Association (CSTA), New York, New York.*

Tinapple, D., Sadauskas, J., & Olson, L. (2013). *Digital culture creative classrooms (DC3): teaching 21st century proficiencies in high schools by engaging students in creative digital projects.* In Proceedings of the 12th International Conference on Interaction Design and Children, (pp. 380-383): ACM.

Tucker, A., Deek, F., Jones, J., McCowan, D., Stephenson, C., & Verno, A. (2003). A Model Curriculum for K–12 Computer Science. *Final report of the ACM K-12 task force curriculum committee.*

Wilson, C., & Guzdial, M. (2010). How to make progress in computing education. *Communications of the ACM, 53*(5), 35-37.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.

Using Cognitive Load Theory to select an Environment for Teaching Mobile Apps Development

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

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

Simon
University of Newcastle
simon@newcastle.edu.au

Barry Wilks
Southern Cross University
barry.wilks@scu.edu.au

Abstract

After considering a number of environments for the development of apps for mobile devices, we have evaluated five in terms of their suitability for students early in their programing study. For some of the evaluation we devised an evaluation scheme based on the principles of *cognitive load theory* to assess the relative ease or difficulty of learning and using each environment. After briefly presenting the scheme, we discuss our results, including our findings about which mobile apps development environments appear to show most promise for early-level programming students.

Keywords: mobile apps, programming education, computing education, cognitive load theory

1 Introduction

The teaching of programming is generally situated in the context of some sort of programming language environment. There have been, and possibly still are, courses that teach programming concepts in the abstract, with no writing or execution of code; but when a course involves code writing and execution, it must necessarily carry out these steps in some sort of environment, whether it be a command-line environment with a simple text editor or a comprehensive Integrated Development Environment (IDE).

Programming environments incorporate not only programming language processors but also tools for many ancillary tasks such as editing, debugging, and file management. We suggest that some of these environments may be so complex as to have an adverse impact on learning outcomes. Professional development tools such as Eclipse (www.eclipse.org) and Visual Studio (www.visualstudio.com) incorporate facilities for advanced programming concepts such as code sharing, versioning, profiling, and more, giving them the potential to be overwhelmingly complex for beginning students. It is important to note that such advanced concepts and capabilities are unnecessary for the purpose of teaching novices introductory programming concepts and skills. These are not only extraneous to the task of teaching and learning introductory level programming, but may well be distracting to learners' focus of attention, overloading their cognitive resources and reducing the capacity of their cognitive processes for learning.

On the other hand there are environments specifically designed for teaching purposes, such as Alice (www.alice.org) and BlueJ (www.bluej.org), which provide the essential tools for learning application development. Between these extremes there are tools such as LiveCode (livecode.com) and App Inventor (www.appinventor.org), which appear to be designed for ease of use, but also to suit continued use by more experienced programmers.

The choice of a programming environment for a particular course hinges upon many factors (Mason & Cooper, 2014; Simon & Cornforth, 2014), including

- the programming language to be used
- the desire to give students experience in a professional development environment
- the availability of teaching aids such as textbooks
- the personal preferences and expertise of the people designing and teaching the course
- cost to students and/or the institution
- access to suitable hardware
- suitability for the purpose of teaching and learning

In this paper we focus on the last of these criteria, pedagogy. In a recent survey of Australian and New Zealand computing academics (Mason and Cooper, 2014), this was the top ranked criterion for selection of a programming language for teaching an introductory programming course, and one of the top four reasons for choosing a development environment.

In considering a possible course in programming apps for mobile devices, we have investigated a number of relevant environments in terms of their usability, including both ease of learning and ease of subsequent use. In this paper we concentrate on the usability of programming environments without considering the many other factors that must contribute to the choice. We go further by developing a method for evaluating usability based on cognitive load theory (CLT) (Sweller, 1994; Sweller, Ayres & Kalyuga, 2011). This method is then applied to several programming environments that were available at the time of writing, and the results discussed.

Other researchers have evaluated IDEs in various ways. For example, Dujmović and Nagashima (2006) used a highly quantitative approach to compare three Java

IDEs, but from the perspective of professional developers. And Kline and Seffah (2005) survey a number of projects that used interviews, questionnaires, or observation to evaluate particular IDEs. However, we are not aware of any prior work using a quantitative approach based on cognitive load theory to assess and compare IDEs.

2 Cognitive Load Theory

2.1 Human cognitive architecture

Humans are limited to a working memory capacity of about seven items (Miller, 1956). Miller observed that this capacity was effectively standardised across our senses. Irrespective of whether the mode of presentation was visual, auditory, taste, or smell, people reliably demonstrated a working memory capacity of 7 (plus or minus 2) for unrelated, 'random' items of information (or stimuli). This was an important observation because it suggested that our capacity to perceive and process the world around us is channelled through a central executive, associated with consciousness that is strictly bounded and relatively small, at least compared to our long-term memory store.

2.2 Expert performance

The kernel of cognitive load theory lies in the argument that the architecture of human cognitive processes, with its limited working memory capacity, may be easily overloaded, whereupon cognitive performance will falter (Sweller, 1988).

There is an apparent contradiction in this when one considers the expert performance of people such as the readers of this paper in the area of computer programming. As you work in an IDE, surely you are attending to and processing and organising and manipulating and coordinating many more than just seven items of information associated with sequences, selections, iterations, objects, variables, functions, properties, methods, and so on.

There are, however, some important riders to this. You do not perceive all of these items as random, disassociated items, but rather, as deeply intertwined and interacting. You have acquired a vast knowledge base regarding the area of programming concepts and skills, and so you are able to effectively work around the limitations of working memory because each of the items that you attend to and process is in fact a highly complex array of information that could be unpacked and deployed into many constituent components. This is a key feature of expertise: you have developed complex schemas that hold well defined, hierarchically structured organisations of knowledge (Chi et al, 1982).

The second key feature of expertise is that experts can attend to, and process, activities in their area of expertise with very low levels of conscious attention (Kotovsky et al, 1985). This is akin to being able to perform on 'automatic pilot', and the terms 'automation' and 'automaticity' have been used to reflect this (Cooper and Sweller, 1987; Shiffrin and Schneider, 1977).

In contrast, novice learners in programming lack both the schemas and their consequent automation that are held by experts. Novices, when seeking to attend to the same information as the expert, must carry it as many more, smaller, packets of information. The fact that they are smaller in size does not help their cause. It is the *number* of elements that is critical, and for novices, this number will probably approach, and possibly exceed, their critical threshold level of cognitive capacity.

2.3 Sources of cognitive load

Cognitive load stems from three sources: intrinsic, extraneous and germane (Sweller, 2010). *Intrinsic cognitive load* is that load imposed by the inherent complexity of the material to be learnt. This is highly dependent upon the level of element interactivity between the individual elements of the information to be learnt, rather than the numerical count of elements per se (Chandler and Sweller, 1991; Sweller et al, 1990). For example, manipulating an array within a loop will impose a higher intrinsic cognitive load than assigning a value to a variable. A common teaching practice is to work from tasks with low levels of element interactivity, and thus low levels of intrinsic load, to those with higher levels of element interactivity, and thus higher levels of intrinsic cognitive load.

Extraneous cognitive load is the load imposed by the way in which information is presented, and depends upon the format of instructional materials and the nature of student activities (Ayres & Sweller, 2005). In the context of teaching programming, extraneous load will also be imposed by the interface that the student is required to navigate in undertaking the instructional materials and learning activities.

Germane cognitive load is the load that occurs as a result of the learners' conscious focus of attention to deliberately remember and understand the learning material (Paas & Van Merriënboer, 1994). That is, germane cognitive load is applied to the actual process of learning.

These three sources of cognitive load are additive, and combine to produce a total cognitive load for each instant of time during an instructional event or learning activity. If, at any point in time, the total cognitive load exceeds the capacity of cognitive resources, then by definition, some aspects of information being attended to must be dropped from consciousness; comprehensions will be lost and learning will be impeded.

Cognitive load theory posits that while the intrinsic complexity of a task remains fixed, the extraneous load may be reduced through re-engineering the instructional materials and/or the learning activities (Sweller et al, 2011). With extraneous cognitive load thus reduced, the released cognitive resources may be re-allocated to the germane aspects of schema acquisition and automation, thus facilitating learning (Paas et al, 2003).

Researchers have identified several specific instructional design principles based on cognitive load theory and have empirically demonstrated their effectiveness. These principles include the worked examples effect (Sweller & Cooper, 1985), the goal free problem effect (Ayres & Sweller, 1990), the split attention effect (Chandler & Sweller, 1991), the redundancy effect (Chandler & Sweller, 1991), the modality effect (Tindall-Ford, Chandler & Sweller, 1997), and the expertise reversal effect (Kalyuga, Ayres, Chandler & Sweller, 2003).

2.4 IDE as a source of cognitive load

Cognitive load theory specifically addresses situations where students are tasked with *learning*. Learning requires cognitive processes to attend to information in working memory, then to organise this new information and manage its transmission to long-term memory, where it will become embedded and organised into existing knowledge, evolving as an increasingly complex network of schemas. The limitations of working memory can become a bottleneck, constricting the interplay of information between the various memory stores.

To make matters worse, the cognitive resources required for the learning process need to actively compete with the demands placed upon resources for attending to and processing other matters. For novices working on a programming task, this will include attending to the relatively many conceptual items of information associated with programming, along with the means of accessing and implementing them by way of components of the IDE.

The instructional design considerations for teaching and learning programming will thus need to consider the extent to which the organisation and presentation of tools in an IDE either increase or decrease the extraneous cognitive load associated with accessing and implementing programming concepts and tasks.

2.5 Assessing cognitive load

The cognitive load experienced during a learning transaction is often assessed by means of a questionnaire or similar instrument. An early instrument was that of Paas (1992), which has formed the basis for numerous studies (Paas et al, 2003). Morrison et al (2014) propose a version specific to computer programming, and present a preliminary report on its use to assess the cognitive load of lectures in an introductory programming course.

If a lecture is found to entail a high cognitive load, it can generally be redesigned to reduce that load. However, this scope for redesigning does not apply to programming environments, which are essentially fixed and invariant. With such environments, the instructor who is aware of cognitive load theory would want to choose an existing environment that offers a low cognitive load when used for the types of task that are typically undertaken by novice students. To this end, we propose a means of assessing the cognitive load associated with programming environments, and apply it to a number of development environments for mobile apps.

When designing teaching and learning resources, teachers should consider the extent of the students' prior knowledge in the content domain. The selection of a development environment will depend on many elements of prior knowledge, for example of the

- environment's programming languages
- target device hardware
- target operating system
- environment's operating system
- general programming concepts

This is a baseline that must be determined when evaluating the suitability of an environment. In the method described in the following section we will assume that this baseline has been clearly accepted and already considered when selecting products for evaluation. This will ensure that consideration and evaluation of the cognitive load factors will be normalised across the evaluation of different products with respect to the level of prior knowledge.

3 Mobile App Development Environments

Just a few years ago it seemed reasonable when discussing the infrastructure for a mobile app development course to consider just one option for Apple iOS devices and one option for Android devices. Goadrich and Rogers (2012) did this, considering XCode and Eclipse. Other platforms and environments were acknowledged, but these two were considered to offer sufficient coverage of the field.

Since then many more development environments have appeared, and the Windows phone has started to make some inroads on the market. The choice of development environments and programming languages is no longer so straightforward.

In the first instance we identified 15 environments that might be worth considering as the basis for an early course on developing mobile apps. This might be a first programming course, or it might be a first mobile apps development course following a more generic introductory programming course. We had decided that we were interested in the development of native apps rather than Web apps; that is, the apps should run directly on the device rather than through a browser.

Initial exploration of these environments led us to narrow the field to just five; in the following subsections we explain our reasoning.

3.1 Environments discarded as too simple

A number of environments appear to be designed for non-programmers, to the point where we did not consider them useful for teaching programming. Having established this, we did not further investigate these environments.

BuzzTouch (www.buzztouch.com) is a medium for designing screens using interface elements and preset behaviours. The actual code generation is carried out in another environment.

Socialize appmakr (www.appmakr.com), Infinite Monkeys (www.infinitemonkeys.mobi), and Orbose (orbose.com) are all menu-based applications with limited functionality, and do not appear to facilitate 'coding' as it is generally understood in computing education.

3.2 Environments discarded as too complex

Several environments appear very much targeted to the professional developer. These environments might well be suitable for students at higher levels of study, but we considered that they would prove too daunting for novice programmers. Indeed, even to install some of them was a major undertaking, requiring multiple reboots and the interpretation of enigmatic error messages. This is something that we would prefer to avoid with beginning students.

Netbeans with Google Android plugin, Eclipse with Google Android libraries, and IntelliJ IDEA with Google Android libraries (www.jetbrains.com/idea), all fell into this category of professional development environments in which there were problems installing either the environments themselves or the supplementary libraries.

Telerik Icenium (www.icenium.com) was ruled out of consideration because it appears to rely on existing advanced HTML5/CSS/Javascript skills, which we cannot assume novice programming students will have.

3.3 Environments discarded for other reasons

GameSalad (gamesalad.com) is designed specifically for writing game apps as opposed to general app programming.

AIDE (www.android-ide.com) was discarded because its code must be written on an Android device: this environment offers no way of writing code on a computer and transferring it to the device. This renders it unsuitable unless we can be sure that every student in the class will have access to an Android device.

3.4 Environments selected for further study

Having eliminated the environments listed above, we were left with just five environments for further investigation.

Visual Studio is an environment used in more than 15% of introductory programming courses in Australia and New Zealand (Mason & Cooper, 2014). **Visual Studio Express for Windows Phone** is a recent variation that permits development on a Windows computer of apps for a Windows phone.

App Inventor (appinventor.mit.edu) is a web-based environment that is used to develop apps for Android phones. In a similar way to Scratch (Maloney et al, 2010), code is built from jigsaw-like code-snippet blocks by dragging them to an editing screen and fitting them together.

TouchDevelop (www.touchdevelop.com) is a web-based environment designed to develop apps for Windows phones, and has recently been extended to include Android devices. Unlike App Inventor, TouchDevelop has a textual form for its code; but because the language was designed to be programmed from smart phones, most of the text entry is carried out by tapping screen buttons rather than from a character-based keyboard.

LiveCode (livecode.com) runs on a Windows, Macintosh, or Linux system and produces mobile apps for Apple and Android devices, as well as desktop applications. LiveCode is a more traditional text-based language, with code entry from a normal keyboard. While LiveCode is designed for writing mobile, desktop, and server applications, its current promotion appears to be aimed primarily at the development of mobile apps.

Xamarin Studio (xamarin.com) runs on a Windows computer to develop apps for Apple, Android, and Windows devices, using C# in an IDE somewhat similar to that of Visual Studio.

Of these environments, App Inventor, TouchDevelop, and Xamarin have been designed specifically for mobile applications development. In the case of Visual Studio a specific version was available for mobile development (Visual Studio Mobile 2012) at the time of evaluation.

All five of these environments are undergoing rapid change at the time of writing. In the rest of this paper we shall report on the environments as we found them in the first half of 2014, expecting that aspects of them will have changed, perhaps substantially, by the time this paper is published.

3.5 Other considerations

There are many reasons why a specific mobile application development environment may be chosen for teaching. These include

- cost to students
- relevance to industry
- number of phone features (eg gyroscope, camera, phone book) available via the environment

Table 1 shows a comparison of these features, and more, for our five chosen environments.

4 Assessing Cognitive Load in Mobile App Development Environments: Method and Application

We set out to investigate the usability of these five environments, with the goal of assessing how suitable each might be as an environment for teaching mobile app development to reasonably inexperienced programming students. We devised a four-step process for the evaluation of the programming environments:

1. choose a selection of small problems whose solutions offer coverage of various aspects of the target environments;
2. record a video screen capture and verbal narration of an experienced teacher solving each problem;
3. view and evaluate the recordings using the cognitive load theory factors discussed below;
4. analyse and compare the evaluations to determine a scoring and ranking of the products studied.

We will describe our method in a general sense as we believe that it will be useful for any instructor selecting a programming environment for any development task. At the same time we will work through our application of the method to the specific task of choosing suitable environments for developing mobile apps.

4.1 Selection of problem set

A programming environment usually contains a rich source of components or tools for development. To evaluate the complexity of an environment, problems were chosen to use small but distinct sets of individual components. Although it would be possible to devise problems that exercise large numbers of components, the interaction between components would increase complexity and complicate the final analysis, so we chose tasks that exercise as few components as possible.

The task descriptions do not have to be strict. The aim is to observe and analyse the use of components of the development environment. Minor variations in interpreting the task will not significantly restrict analysis of its presentation and use. Similarly, while different problem solvers might chose different solutions to the problem, all will entail the targeted environment component.

For example, in targeting mobile apps development environments, we considered three distinct areas. First there is the coding itself. Second, these environments tend to incorporate a separate graphical area for designing the user interface. Finally, different environments will have different ways of managing external media, which are of high importance in mobile apps. So for our evaluation of mobile development environments we chose three separate tasks that together exercise the coding, layout, and media aspects of the environments.

Task 1: Hello world program

This task requires the developer to create an application that displays a "Hello, world!" message when a button is clicked. This was considered to be the simplest mobile application that requires processing of user actions, and that does not correspond to a default application provided in any of the development environments. A key facet of this task is the ability to create a testable application using the environment's application building tools.

Task 2: Animal display

This task requires the display of four animal images as chosen by a selection widget. The selection widget type is not specified, as this might vary with environment and target platform, but the developers were expected to use the simplest possible widget for this task. The animal images are to be in common graphic formats. This problem exercises the environment's media processing ability, in this case with images. The selection widget also had to be more complex than a simple button, in order to cater for the four-way choice.

Task 3: Hello world permute

This task requires the display of permuted versions of the string "Hello, world!" on the press of a button. This task exercises the programming language part of the environment by requiring a small but non-trivial text-manipulation computation to be programmed. This task has previously been used by Goadrich and Rogers (2012) to compare two environments, and by Simon and Cornforth (2014) to further compare those with a third environment.

These tasks were selected on the basis that they would progress from the simplest possible task, through one with a little more complexity in both the graphical user interface and the coding, to one in which the algorithm and the coding might appear fairly complex to a novice programmer.

4.2 Recording the solution

Solutions to the problems identified above were recorded by screen capture software, with the recording including a think-aloud narration to help the evaluators understand why particular actions are taken during the process.

Three of the four authors were each allocated one or two environments, in which they undertook all three tasks, producing 15 screencasts in total.

Some screencasts included steps that are not strictly necessary for completion of the task, such as changing the default names of interface objects. These steps were not counted as part of the activity or included in the evaluation. We also excluded any debugging steps apart from the standard steps required to build and test the final product. Our goal was to compare the environments themselves, and for this we required a straightforward bug-free coding of the simplest solution for each activity.

It would not have been appropriate to have students make the screencasts. Firstly, any problems with the use of the IDE would then be conflated with the students' learning problems. Second, such an approach would entail having students use five different IDEs with different programming languages, which would be a somewhat unusual approach to selecting the IDE and language for a course.

4.3 Evaluation of the recordings

Analysis of each screencast began by iteratively breaking

Table 1: big-picture considerations of the environments

Environment	**App Inventor**	**Touch-Develop**	**Live-Code**	**Xamarin**	**Visual Studio**
Cost to students	free	free	free	free	free
Visual cues	yes	yes	yes	yes	yes
Visual debugger	no	yes	yes	yes	yes
Graphical user interface	yes	yes	yes	yes	yes
Difficulty of installation	low	low	medium	high	medium
Cross-platform development	no	yes	yes	yes	no
Relevant to industry	yes	yes	yes	yes	yes
Open source	no	no	yes	no	no
Available support material	high	high	medium	high	medium
Can port to more than one platform	no	yes	yes	yes	no
Number of files user has to manage	0	0	1	many	many
Degree of textuality (block-based → typed code)	low	medium	high	high	high
Phone features available via the environment	high	high	medium	high	high
Required prior knowledge . . .					
Procedural algorithms	yes	yes	yes	yes	yes
Object orientation	no	no	no	yes	yes
GUI widgets and event processing	no	no	yes	yes	yes
Target mobile operating system	no	no	no	yes	yes
Specific programming language	no	no	no	yes	yes

the program development into a series of steps. The same task will usually require different numbers of steps in different environments, and steps that appear in more than one environment may vary in complexity.

To evaluate the steps we devised a series of cognitive load factors that are either directly observable from the screencasts or readily deduced; see Table 2. While these factors might vary according to the activity being addressed, we posit that they are applicable to a broad range of software development tasks.

Table 2: cognitive load factors

CLT Factor	Description
Factors that add to cognitive load; scored as low/medium/high	
EC: Environment schema complexity	Breadth/depth of environment and/or language-specific schemas that are required to perform this step
PC: Programming schema complexity	Breadth/depth of general programming schemas that are required to perform this step
TB: Think back	Number of elements from previous steps needing to be kept in mind to perform this step
I: Interactivity	Complexity of the interactions between environment schemas, programming schemas, and think back required to perform this step
PE: Relevant physical elements	Number of relevant physical elements appearing on screen that may be chosen as part of performing this step
D: Distractors	Number of physical elements in view but irrelevant to performing this step
WP: Windows/ palettes	Number of windows/palettes that are visible and active on screen while performing this step
SA: Split attention source	Extent of physical separation between elements of information or interaction that need to be mentally integrated to order to perform this step
Factors that can reduce cognitive load; scored as present/absent/NA	
PH: Prompts/hints	Instructions viewable in text or graphical form for performing this step
GS: Guiding search	Attention drawn to next element required for performing this step; for example, by highlighting text instructions or target entry field
CS: Context-sensitive help	Help available as scaffold for performing this step; for example, tool tips or other prompts indicating the purpose of an element
G: Groupings	Clustering of elements into related functionality associated with performing this step; for example, automatic indentation, clustering of menu items

We then examined the screencasts in detail, assessing each step according to each evaluation criterion. This resulted in 15 two-dimensional tables, one for each screencast, with rows representing the steps and columns representing the CLT factors.

Finally we describe the scoring system. As indicated in Table 2, factors that add to cognitive load were rated as low, medium, or high, while factors that can diminish cognitive load were rated as present, absent, or not applicable. These ratings were all assigned numerical values, as shown in Table 3.

The scoring scale in Table 3 is obviously non-linear. This is consistent with CLT theory in that a difficult step, imposing high cognitive load, will impact considerably more upon cognitive resources than a simple step that entails lower cognitive load, and might even block progress completely. Given this effect, the choice of the value 4 rather than, say, 5 or 10 is discretionary, but has proven useful for our analysis in these mobile app development environments. Likewise, we note the presence of a cognitive load reduction factor by subtracting 2 from the score for the step. Again, this judgement is discretionary, and is based on the supposition that a factor that reduces cognitive load would typically offset a medium-level factor that increases cognitive load. Both of these factor levels require a level of mindfulness from the user rather than just an acknowledgment of their existence.

The evaluations were arrived at by consensus. Two of the authors jointly evaluated the recordings and then allocated an agreed level (low, medium, or high for loading factors, and present, absent, or NA for the reduction factors) for each entry in the Environment/CLT Factor tables. The other authors checked the evaluations and initiated discussion on any values they were not in agreement with.

There was no assessment of the inter-rater reliability of the method, essentially because of the time constraints in choosing the IDE for a proposed course. Future work would certainly include checking the inter-rater reliability.

A highly reduced example table is shown in Appendix 1. This includes all of the steps in the hello world task in App Inventor, one row for each step; but because of the limited space, only a small sample of the columns, showing five of the load-adding factors and three of the load-reducing factors.

4.4 Analysis of data

In this section we suggest ways of evaluating the data gathered in the previous section. Bearing in mind the overall goal of ranking a number of environments, some standard summative statistics can be applied to each program development task and comparisons made

Table 3: step/factor score scale

Score	Description
1	low – minimal or no cognitive load
2	medium – requires consideration
4	high – substantially present
2	reduction factor present (subtracted)
0	reduction factor not present
0	reduction factor not relevant to this step

between environments. The same measures can also be applied to all three tasks combined, regarding them as a single, more complex, task. Here are the measures that we have devised.

Number of steps: this is the total number of steps required to complete the task in the development environment. It does not consider the difficulty of each step.

Cognitive load score per step (CLSS): the cognitive load score for a step is calculated as the sum of the numeric values of the step's individual CLT factors using the scoring system in Table 3.

The measure is determined by:

$$CLSS = EC + PC + TB + I + PE + D + WP + SA - (PH + GS + CS + G).$$

This measure is important because a step with a high CLSS is likely to overload novice programmers and hence block progression on the programming activity.

Minimum and maximum CLSS: the maximum CLSS over all the steps involved in carrying out the task can be used as a measure of the expertise required to use this environment for this activity. The minimum is provided for completeness.

Mode and median CLSS: the mode and median of CLSS across all steps allow a comparison of the central tendencies of CLT difficulty of each step between environments and between tasks.

Threshold score: this is the proportion of CLSS scores above a threshold value intended to represent high cognitive load for a particular cohort of learners. After examining the scores of each task in each environment, we chose a threshold value of 10 as indicating a step with a relatively high cognitive load *for novice learners*. The proportion of steps with a CLSS over this value is therefore indicative of how much of the task is cognitively taxing for this cohort in each environment.

Average cognitive load per factor (ACLF): averaging the scores for each individual cognitive load factor across all steps allows the relative contribution of various cognitive load factors to be determined. For example, the ACLF for think back (TB) for a particular environment would be calculated by the formula $ACLF_{TB} = sum(TB) / steps$. ACLF will always be between the low and high scores given for each factor. For the scoring that we have used (Table 3), ACLF will be between 1 and 4 inclusive.

Proportion of steps assisted (PSA): for each cognitive load factor whose presence reduces cognitive load (Table 2), this is the proportion of steps in a task that are assisted by that factor, and is therefore something of an offset to ACLF. The PSA for a factor is calculated by the count of the steps in which the reduction factor was present, divided by the number of steps. *PSA = count (reduction factor present) / steps*

Table 4 shows the number of steps, CLSS measures and threshold score for each task in each environment, ordering the environments by increasing number of steps for each task. The table clearly shows how the number of steps and the complexity of steps must be considered together. For example, in the hello world task, LiveCode scored lowest in both the number of steps and the proportion of steps that exceeded our threshold, indicating that it presented the least cognitive load for novice users. However, in the animal display task, Visual Studio presented the lowest number of steps but the highest proportion of steps that exceeded our threshold. This indicates that even though there were fewer steps, a clear majority of the steps would provide excessive cognitive load to a novice user.

Table 5 shows the average cognitive load per factor, highlighting the highest value of each factor for each task. This table allows us to judge the average cognitive load per CLT factor, independent of the number of steps. For example, in the string permute task, Visual Studio provided the highest average cognitive load per step for the think back (TB) factor (2.06), while LiveCode provided the lowest (1.25). This supports our intuition that Visual Studio requires a much higher degree of memory of previous actions than LiveCode, which provides a larger amount of contextual information to its user. In contrast, for the same task, LiveCode provided the highest average environmental schema complexity (EC) factor and Visual Studio provided the smallest. This supports our intuition that more complex algorithms can be more concisely expressed in a traditional programming language than in the more verbose programming

Table 4: measures of cognitive load per step

		CLSS				Threshold
1: Hello World	**STEPS**	**Min**	**Max**	**Mode**	**Median**	**Score**
App Inventor	18	2	10	9	8.5	**6%**
LiveCode	13	5	8	7	8	**0%**
Touch Develop	22	2	13	8	7	**14%**
Visual Studio	18	4	13	8	8	**28%**
Xamarin	13	8	13	8	10	**54%**
2: Display Animals						
App Inventor	61	2	16	8	9	**41%**
Livecode	54	5	16	8	8	**20%**
Touch Develop	61	2	13	4	6	**13%**
Visual Studio	48	4	26	16	15	**83%**
Xamarin	50	6	26	11	11	**64%**
3: Permute						
App Inventor	60	2	16	9	10	**52%**
LiveCode	20	5	12	8	8	**30%**
Touch Develop	88	2	17	8	8	**22%**
Visual Studio	36	4	23	15	15	**83%**
Xamarin	26	7	27	8	10	**58%**

Table 5: average cognitive load of each factor, highlighting the highest value for each task

	ACLF (Average Cognitive Load per Factor)							
1: Hello World	**EC**	**PC**	**TB**	**I**	**PE**	**D**	**WP**	**SA**
App Inventor	1.33	1.11	1	1	***1.56***	1.5	***1.39***	1
LiveCode	1.23	1.08	1.08	1	1.08	1.54	1.31	1
Touch Develop	***1.64***	1.27	1.14	1.18	1.41	1.41	1	***1.32***
Visual Studio	1.33	1.28	1.11	1.11	1.11	***1.72***	1.22	1.06
Xamarin	1.62	***1.62***	***1.23***	***1.31***	1.46	1.38	1.15	1.15
2: Display Animals								
App Inventor	1.56	1.44	1.46	1.51	1.57	1.2	1.16	1.07
Livecode	1.54	1.28	1.24	1.26	1.35	1.59	1.35	1.19
Touch Develop	1.49	1.46	1.3	1.36	1.74	1.05	1	1.15
Visual Studio	***1.96***	1.46	1.58	1.38	***2.6***	***3.29***	***3.19***	1.71
Xamarin	1.76	***2.34***	***2.14***	***2.1***	2.54	2.3	2.08	***1.74***
3: Permute								
App Inventor	1.37	1.85	1.48	1.68	1.48	1.32	1.18	1.17
LiveCode	***1.5***	1.3	1.25	1.4	1.5	1.6	1.65	1.05
Touch Develop	1.4	1.77	1.53	1.67	1.59	1.65	1.01	1.43
Visual Studio	1.19	1.94	***2.06***	1.97	***2.64***	***3.44***	***3.11***	1.44
Xamarin	1.38	***2.19***	1.85	***2.04***	2.19	2	1.65	***1.5***

language of LiveCode. Both Visual Studio and Xamarin scored highly on relevant physical elements (PE), distractors (D) and windows/palettes (WP). For more experienced programmers with developed schemas about programming and the environment, having a large range of palettes and options available on screen will be an advantage as all options have easier access. For novices, the availability of options and palettes provides extra cognitive load which may add to other load and impede learning.

Table 6 shows the proportion of steps assisted by CLT factors which reduce cognitive load. This is useful to consider separately because it provides a judgement on how explicit design choices of the IDEs actually contribute to cognitive load reduction during solution of the chosen problem set. As an overall comparison it can be seen from this table that Touch Develop used all four factors to better reduce cognitive load in all three problems. It is also interesting how the contributing factors varied according to problem type. For example, the string permute task, which required the user to design an algorithm, showed App Inventor providing the lowest amount of cognitive load reduction in its grouping factor. This was not the case for this IDE in the hello world task, where it provided the highest contribution to cognitive load reduction from its grouping factor. This suggests that App Inventor's grouping design was aimed at facets of mobile app development other than the algorithm design.

As a final step we computed the average threshold score of each environment over all three tasks. While an instructor trying to choose between these environments should carefully examine all of the measures, this average has the advantage of being a single measure for an environment, thus permitting a very quick high-level comparison of the environments. The computed averages are as follows:

• TouchDevelop:	16%
• LiveCode:	17%
• App Inventor:	33%
• Xamarin:	59%
• Visual Studio:	65%

Table 6: proportion of steps assisted by each load-reducing factor, highlighting highest values

	PSA (Proportion of Steps Assisted)			
1: Hello World	**PH**	**GS**	**CS**	**G**
App Inventor	11%	11%	6%	***78%***
LiveCode	54%	8%	8%	54%
Touch Develop	***64%***	***50%***	***18%***	50%
Visual Studio	11%	17%	0%	50%
Xamarin	15%	0%	0%	38%
2: Display Animals				
App Inventor	23%	3%	5%	51%
Livecode	19%	7%	22%	50%
Touch Develop	***74%***	***38%***	38%	67%
Visual Studio	11%	17%	0%	50%
Xamarin	50%	10%	***48%***	***74%***
3: Permute				
App Inventor	30%	10%	2%	47%
LiveCode	30%	10%	45%	***75%***
Touch Develop	***72%***	***31%***	***50%***	69%
Visual Studio	36%	3%	28%	67%
Xamarin	35%	0%	35%	69%

These average scores show a clear clustering of results, which will be discussed in the following section.

5 Discussion and Conclusion

In this paper we have addressed the usability criterion for selecting an environment for teaching programming to relative novices. We have introduced a method for evaluating and comparing usability among a selection of candidate products, and have applied this method to the selection of a programming environment for teaching the development of apps for mobile devices.

The mobile development environments that we evaluated fell clearly into three groups. TouchDevelop and LiveCode, with threshold scores of less than 20%, permitted the development of code with the least relative cognitive load. Despite the fact that it was designed for, and is typically used for, novice programmers, App Inventor had double the threshold score of these two environments, indicating a substantially higher relative cognitive load. Both Visual Studio and Xamarin Studio had threshold scores of around 60%, nearly double again, indicating another substantial leap in the relative cognitive load required to develop mobile apps in these environments.

It would be incorrect to conclude that all we have achieved here is to confirm an intuitively obvious result. While different people have different intuitions, we nevertheless imagine that many readers would expect the block-based App Inventor to impose the least extraneous cognitive load, and that is not what we have found. Things that are "obviously" the case sometimes turn out to be incorrect. The tool and methodology presented here is a movement towards enabling objective analysis and comparisons between dissimilar IDEs using Cognitive Load Theory.

Readers interested in applying this method to their own selection of programming environments should remain aware that this single figure is derived from many conflicting factors, and that each factor should be considered in its own right before a decision is made. For example, Table 4 shows that TouchDevelop typically requires more steps than the other environments to carry out the same task. Because coding in this environment is carried out by way of tapping buttons on the screen, coding a loop might be counted as half a dozen separate steps, whereas coding the equivalent loop in a keyboard-based environment might count as a single step. On the other hand, Xamarin and Visual Studio tend to score quite high on cognitive load score per step.

Therefore it does not necessarily follow that early courses in mobile apps development should choose between TouchDevelop and LiveCode. There are many factors involved in selection of a program development environment, and instructors should consider all of the factors that pertain to their circumstances. However, it does follow that if the other factors are more or less equivalent, one of these two environments might be a good choice, especially if targeting novice programmers.

App Inventor has been used for at least one university-level course (Robertson, 2014), but only at the outset, with a move during the course to a more traditional development environment. We tend to concur with Robertson that App Inventor is in one sense too simple and in another sense too frustrating to form the basis for a full university course.

Visual Studio or Xamarin Studio might be chosen for a mobile apps development course that is not the first programming course, particularly if the same environment had been used for the introductory course, or if the students were already familiar with the programming language chosen. They might also be chosen if the instructors particularly wanted to introduce the students to these environments. However, instructors should be alert to the substantially higher cognitive loads imposed by these environments, and should be prepared to scaffold their novice students appropriately.

The design and application of the cognitive load analysis method described here offers a prospective way for instructors to assess different IDEs in a wide range of contexts. The method is open to modification with respect both to the cognitive load factors that are considered and to the calibration of scoring used to assign values.

6 References

P. Ayres and J. Sweller (1990). Locus of difficulty in multistage mathematical principles. *American Journal of Psychology*, 105(2):167-193.

P. Ayres and J. Sweller (2005). The split attention principle in multimedia learning. In *The Cambridge Handbook of Multimedia Learning*, ed. Richard Mayer, Cambridge University Press, 135-146.

P. Chandler and J. Sweller (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8:293-332.

M. Chi, R. Glaser, and E. Rees (1982). Expertise in problem solving. In *Advances in the Psychology of Human Intelligence*, Erlbaum, Hillsdale, NJ, 7-75.

G. Cooper and J. Sweller (1987). Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79:347-362.

J. Dujmović and H. Nagashima (2006). LSP method and its use for evaluation of Java IDEs. *International Journal of Approximate Reasoning*, 41:3-22.

M.H. Goadrich and M.P. Rogers (2012). Smart smartphone development: iOS versus Android. *ACM SIGCSE Technical Symposium (SIGCSE'12)*, 607-612.

S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller (2003). Expertise reversal effect. *Educational Psychologist*, 38:23-33.

R. Kline and A. Seffah (2005). Evaluation of integrated software development environments: challenges and results from three empirical studies. *International Journal of Human-Computer Studies*, 63:607-627.

K. Kotovsky, J.R. Hayes, and H.A. Simon (1985). Why are some problems hard? Evidence from tower of Hanoi. *Cognitive Psycholology*, 17:248-294.

J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):16.

R. Mason and G. Cooper (2014). Introductory programming courses in Australia and New Zealand in 2013 – trends and reasons. *16th Australasian Computing Education Conference (ACE2014)*, Auckland, New Zealand, 139-147.

B.B. Morrison, B. Dorn, and M. Guzdial (2014). Measuring cognitive load in introductory CS: adaptation of an instrument. *Tenth International Conference on Computing Education Research (ICER2014)*, Glasgow, Scotland, 131-138.

F.G. Paas (1992). Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *Journal of Educational Psychology*, 84(4):429.

P. Paas, A. Renkl, and J. Sweller (2003). Cognitive load theory and instructional design: recent developments. *Educational Psychologist*, 38(1):1-4.

F. Paas, J.E. Tuovinen, H. Tabbers, and P.W. Van Gerven (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist*, 38(1):63-71.

F.G.W.C. Paas and J.J.G. Van Merriënboer (1994). Variability of worked examples and transfer of geometrical problem-solving skills: a cognitive-load approach. *Journal of Educational Psychology*, 86:122-133.

J. Robertson (2014). Rethinking how to teach programming to newcomers. *Communications of the ACM*, 57(5):18-19.

R. Shiffrin and W. Schneider (1977). Controlled and automatic human information processing II. Perceptual learning, automatic attending and a general theory. *Psychological Review*, 84:127-190.

Simon and D. Cornforth (2014). Teaching mobile apps for Windows devices using TouchDevelop. *16th Australian Computing Education Conference (ACE2014)*, 75-82.

J. Sweller (1988). Cognitive load during problem solving: effects on learning. *Cognitive Science*, 12:257-285.

J. Sweller (1994). Cognitive load theory, learning difficulty and instructional design. *Learning and Instruction*, 4:295-312.

J. Sweller (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, 22:123-138.

J. Sweller, P. Ayres, and S. Kalyuga (2011). *Cognitive Load Theory*. Springer, New York.

J. Sweller, P. Chandler, P. Tierney, and M. Cooper (1990). Cognitive load as a factor in the structuring of technical material. *Journal of Experimental Psychology: General*, 119:176-192.

J. Sweller and G.A. Cooper (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2:59-89.

S. Tindall-Ford, P. Chandler, and J. Sweller (1997). When two sensory modes are better than one. *Journal of Experimental Psychology: Applied*, 3:257-287.

Appendix: Example evaluation (partial): App Inventor environment with the hello world task. A number of columns have been removed in order to fit the table on this page.

Step	Description	Notes	Prior knowledge	Complexity of schema required	Relevant physical elements	Distractors	Windows/ palettes	Prompts/hints	Guiding search	Groupings
1	Turn off intro help screen		low	low	low	low	low	yes	no	yes
2	second help screen close		low	low	low	low	low	no	no	no
3	Create new project		low	low	low	low	low	no	yes	yes
4	Name project		low	low	low	low	low	yes	yes	no
5	Add button	Drag and drop	low	low	medium	medium	medium	no	no	yes
6	Change label of button	in properties palette	low	low	medium	medium	medium	no	no	yes
7	Add textbox	drag and drop	low	low	medium	medium	medium	no	no	yes
8	Rename textbox		low	mediu m	low	medium	medium	no	no	yes
9	Go to blocks	To start editing code	medium	low	low	medium	medium	no	no	yes
10	Go to Button 1	to choose event associated with Button 1	medium	low	medium	low	low	no	no	yes
11	Choose click event	"When Button1 click do .."	low	low	medium	medium	low	no	no	yes
12	Go to Textbox blocks	to choose action associated with Label 1	medium	low	medium	low	low	no	no	yes
13	Choose set text block	"set label1.text to .."	low	low	medium	medium	medium	no	no	yes
14	Go to text blocks	to choose string block	medium	low	medium	low	low	no	no	yes
15	Choose string block	go to text, choose first option (string), drag and drop to right position.	medium	mediu m	medium	low	medium	no	no	yes
16	set value to "Hello World"		low	low	low	medium	low	no	no	no
17	Connect Emulator	Connect menu - emulator	medium	low	medium	medium	low	no	no	yes
18	Test application	click on button in app	low	low	low	low	low	no	no	no
	Counts:	low	12	16	8	9	11	2	2	14
		medium	12	4	20	18	14	11%	11%	78%
		high	0	0	0	0	0			
	Average (weighted):		1.33	1.11	1.56	1.50	1.39			

Student Perceptions of Flipped Learning

David Murray **Terry Koziniec** **Tanya McGill**

School of Engineering and Information Technology
Murdoch University,
90 South St, Murdoch WA 6150
Email: D.Murray@murdoch.edu.au
Email: T.Koziniec@murdoch.edu.au
Email: T.McGill@murdoch.edu.au

Abstract

Flipped learning has been the subject of significant hype and attention but descriptions of the development and the evaluation of this pedagogical model are lacking. Flipped learning is an inverted teaching approach where students learn the basics via short videos at home, then come to class to complete challenges and clarify any misunderstandings. This paper describes how an IT unit was delivered using the flipped learning approach. A survey was used to determine how students perceived flipped learning. Students were generally positive about the approach, particularly the convenience and flexibility of the flipped videos. Although face to face teaching time was reduced in this flipped learning implementation, students felt that they interacted more with their instructors and peers. Students felt strongly positive to walkthroughs and were mixed as to the need for the instructors face. Significant efforts to produce high quality and engaging videos were made, but the survey suggested that students learnt the most during tutorial time. The relative importance of interactive tutorials is congruent with a large body of research and pedagogical approaches advocating the importance of active student-centred learning.

Keywords: Flipped learning, student-centred learning, inverted classroom, online learning, blended learning, IT education

1 Introduction

The flipped classroom pedagogical approach generally involves inverting the typical university style of lecture-based teaching, to get students to view short video lectures at home before the class session, and reserving class time for more interactive activities such as discussions, group exercises or projects. This approach has received a lot of publicity, but there has been little formal evaluation of the impacts on student satisfaction or performance. There has also been little research on how the pedagogical approach can be used in teaching Information Technology (IT).

This paper seeks to address this gap by describing the development and evaluation of a new flipped classroom IT unit called Introduction to Server Environments and Architectures (ISEA). The evaluation was performed to determine student perceptions of the efficacy of flipped learning. Content, accessibility, the amount of face-to-face interaction and preference on video types were all specific areas of interest. The results provide insight into the value of adopting a flipped classroom approach to teaching in IT, and provide understanding about the contribution of different aspects of the design to student satisfaction with their learning.

This paper is structured as follows. Section 2 reviews the literature of online learning, flipped learning and other similar pedagogical approaches. Section 3 describes the unit and the approach taken to create and deliver the videos. The method for the evaluation is detailed in Section 4 and 5 discusses the results. Section 6 concludes the paper.

2 Literature Review

Online learning has received significant attention in the past decade, with increasing amounts of tertiary instruction being delivered online. There are various pedagogical models that can be used to facilitate online instruction. Some courses are delivered purely online, with no face to face interaction. Blended learning is a broad term and simply refers to any learning program where more than one delivery mode is used. Within blended learning, numerous different pedagogical approaches exist including student-centred learning, active learning and problem based learning. Flipped classrooms (AKA inverted classrooms) are one way of implementing active student-centred learning. Flipped learning inverts the traditional approach of teaching the basics in class and reserving practical activities for homework. In flipped learning, the basics are covered in short video lectures which are watched before attending class. This reserves class time for interesting and engaging problem based learning. Any difficulties with the basics can also be identified and addressed during class time.

Flipped learning has received a great deal of popular attention, particularly due to the success of the Khan Academy, which offers a library of over 3,000 videos. The creator, Salman Khan, has been a strong advocate of the flipped learning model.

Despite the flipped learning hype, there is very little evidence about the specific merits of flipped learning and there have been calls for quantitative and rigorous qualitative research on flipped learning (Hamdan 2013, Bishop 2013). In their review of the research on flipped learning, Bishop and Verleger (2013) identified 11 previous studies that have explored student perceptions of flipped learning and concluded that although the results were mixed, with a small proportion of students disliking the approach, students generally had positive perceptions of flipped learning. More recent studies by Butt (2014) and

Kong (2014) have also reported positive student perceptions.

Evidence on the ability of flipped classroom approaches to improve learning outcomes is more limited, but despite this, many of the elements are based on established and well researched learning strategies. The reduced emphasis on traditional lectures is supported by the literature, with a recent meta-analysis of 225 active learning studies in Science Technology and Math (STEM), finding that average examination scores improved by 6% and that students in traditional lecturing classrooms were 1.5 times more likely to fail (Freeman 2014).

Pierce et al (2012) used flipped learning in their pharmacotherapy class and found modest improvements in student performance as well as positive student perceptions that suggested that students recognised the pedagogical benefits and the convenience of the flipped classroom approach. Kong (2014) used a flipped classroom approach in an integrated humanities class and found that students taught in this way significantly increased their domain knowledge.

Within the domain of IT learning and teaching there have only been a limited number of published studies on the success of flipped classroom approaches to teaching IT. Gannod, Burge and Helmick (2008) described on a pilot implementation of a service oriented architecture course which was received very favourably by students. Both Day and Foley (2006) and Davies, Dean and Ball (2013) have taken their evaluation further and reported on learning outcomes. Day and Foley (2006) implemented a flipped classroom intervention for a computer interaction course and found that those students in the flipped classroom group received significantly higher results on both assignments and tests. More recently, Davies, Dean and Ball (2013) also noted improvements in learning for students in the flipped classroom version of a spreadsheet course. They however, identified the short duration of their class (5 weeks) as a limitation and called for further research on the use the flipped classroom approach in IT teaching.

3 Description of the Study

3.1 Information about the unit

A flipped learning approach was used in a first-year first-semester university unit called Introduction to Server Environments and Architectures (ISEA) at Murdoch University. There were 85 enrolled students, of which, 75 were enrolled in internal mode and 10 were enrolled in external mode. This unit introduces students to Linux and Windows operating systems, with an emphasis on servers. The unit also covers virtualization and Amazon EC2 is used as a vehicle to explore cloud computing. The final assignment task involves launching a Linux server in the cloud, linking it to Domain Name System (DNS) and installing/customising a server application such as HTTP. ISEA was a new unit and ran for the first time in Semester 1 2014. As a result comparisons with a traditional, non-flipped version, are not possible.

The familiar activities which accompany many university units were used. A unit guide dictated assessment and the breakdown of topics. A brief abstract was provided to introduce each weekly topic and tie the video, reading, discussion and lab elements together as a cohesive unit. All the elements required for the course were provided as links from the Learning Management System (LMS). Many of the units at the university have a 2 hour lecture and 2 hour tutorial format. In ISEA there was an introductory lecture in week 1, to describe the flipped learning approach, then all subsequent content was delivered online using short 3-20 minute videos.

In flipped learning, the tutorials are designed to be interactive and build upon the basics established in the videos. The ISEA tutorials began with a 20 minute discussion about something topical relating to the unit or the recent videos. While group discussions are the norm in arts degrees, they are rare for applied and technically focused IT units. Following the 20 minute discussions, students completed practical work which built upon the weekly videos.

3.2 Flipped Video Production

When converting a university unit to flipped learning, the new element required for the course is the short videos. The creation and production of these videos is likely to be the most time consuming element for unit coordinators.

3.2.1 Audio

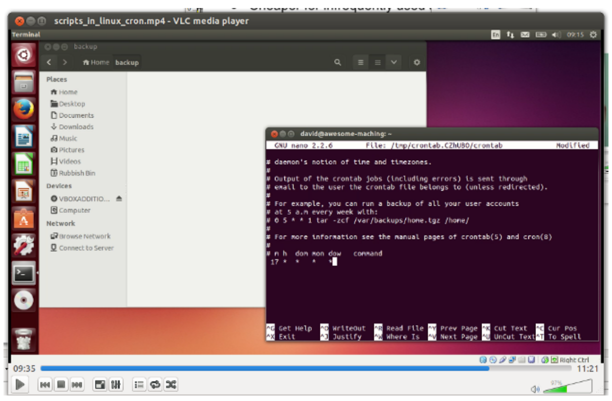
An early decision was made to pursue quality audio in presentations. PCs, tablets and smartphones are all capable of high quality audio, while video quality is heavily dependent on the student's viewing platform, with small screen mobile devices severely limiting the effectiveness of a visual message. The unit coordinators purchased a popular USB omindirectional microphone and a broadcast quality directional microphone. Both were capable of quality audio but their characteristics and usage were quite different.

The USB microphone was suited to presentations where the presenter needs to move around as its positioning was not critical. It also made video presentations more casual as the microphone could be placed inconspicuously. The downside to the USB microphone's ability to capture audio from any position was its susceptibility to picking up background noise. Conversely the broadcast microphone was insensitive to background noise but required positioning in a manner often seen with radio announcers. This made it suitable for "voice-overs" but more difficult to use inconspicuously when combined with video of the presenter's face. Achieving clear audio is not difficult but each technology is optimised for particular conditions and matching the equipment characteristics to the environment was something we found to be important but not obvious at the outset.

3.2.2 Video

A variety of video capture methods were employed. These ranged from basic screen capture applications such as the open source "simplescreenrecorder" and "CaptureMyDesktop" which were used for demonstrating computer based activities and "walk-throughs" of screen based activities. An example of this video type is shown in Figure 1. The 'chalk and talk' approach, where the instructor talked the students through an idea while drawing a diagram or doing some math, was also used. This presentation type is shown in Figure 2 and is similar to the video type used on Khan Academy.

These videos were the most simple to produce as the steps used to combine the video and voice-over are flexible. Both can be captured at once and easily edited later. Alternatively a perfect run-through can be obtained first and then the voice-over can be added later, while the presenter watches the prerecorded action. For instructors seeking to record their



```
  GNU nano 2.2.6          File: /tmp/crontab.CZhUBO/crontab                Modified

# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
 17 *  *   *   *

^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Figure 1: Computer aided demonstration with audio narration

content, voice-over demonstrations are an excellent starting point and introduction to combining video with audio and exploring the basic functions of their chosen video editing environment.

A number of video styles were employed to deliver recorded versions of traditional PowerPoint presentations. One example of this video type is shown in Figure 3. In some instances the presenter's face featured heavily in the recording while in other cases a small face in a window merely reminded viewers who the presenter was. To record the presenters face, internal and external webcams and a digital SLR camera were employed, with each step-up improving the quality of the image. Adding video to the presentations adds considerable complexity. Issues encountered included difficulty in placing cameras in positions that lead to natural looking environments and problems with misaligned audio and video (lip-sync). Editing video without producing jarring and disconcerting jumps in the images is something that requires planning and the unit coordinators found that "delivery" quickly becomes "production". Depending on skill and level of perfectionism, "production" can quickly consume time and creative energy that could otherwise have been devoted to improving the instructional content.

3.2.3 Delivery of videos

The final edited videos were were standardised as high definition 720p in an MP4 container and were uploaded to the university LMS site for students to download. The maximum size of each file was less than 100 MB. Despite testing the files in Windows, Apple and Android environments there were still reports of students experiencing difficulties and the accessibility being less than might be expected from commercial sites such as YouTube.

For some students there were clearly local client issues and quality Internet connections are not universal in Australia. The instructors did find these aspects distracting and time consuming to deal with, particularly as students involved become frustrated with the technology. As the size of the class increases the number of these issues will also grow. There is certainly an incentive to have video content served and managed by a third party that has the expertise and experience to ensure multi-platform compatibility, if those services are not already present in the host's organisation.

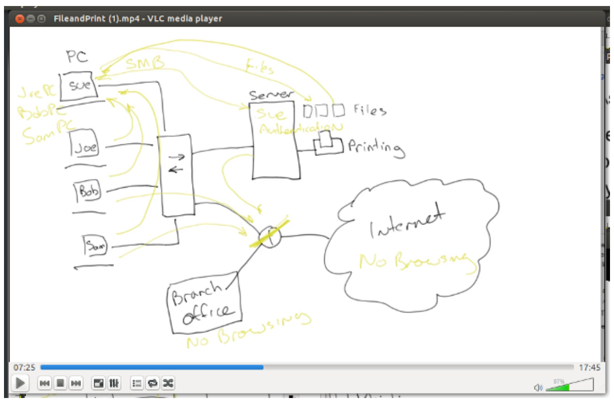
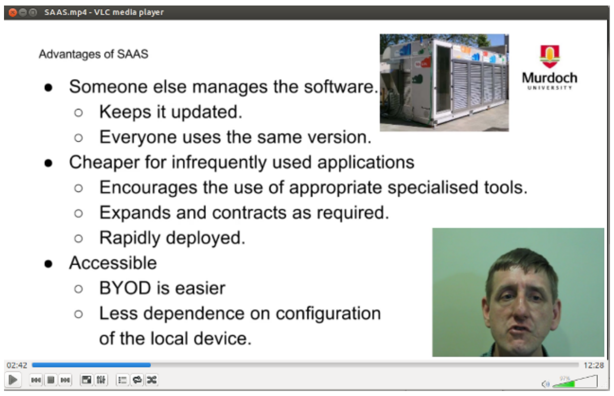


Figure 2: Traditional blackboard style chalk and talk



Advantages of SAAS

- Someone else manages the software.
  - Keeps it updated.
  - Everyone uses the same version.
- Cheaper for infrequently used applications
  - Encourages the use of appropriate specialised tools.
  - Expands and contracts as required.
  - Rapidly deployed.
- Accessible
  - BYOD is easier
  - Less dependence on configuration of the local device.

Figure 3: Talking head and slide show

4 Methods

Student perceptions of flipped learning were measured using an online survey. Internal students were delivered a consent form and an online survey at the beginning of the final tutorial (contact the authors for a copy of the survey questions). The research investigators, who were involved in the unit, did not enter the class while surveys were being completed by consenting students and were not provided access to any survey data until after final grades had been submitted.

Students who do not attend classes on campus and are enrolled in external mode, were emailed information about the evaluation and invited to complete the online survey at a time convenient to them. The survey was approved by the Murdoch University Human Research Ethics Committee (Approval No 2014050). The response rate was 73.6% (56 of the 76 who completed more than 50% of the assessment activities).

5 Results

5.1 Access and flexibility

One of the benefits of flipped learning videos is that the content is accessible via all devices and can be viewed and re-viewed at the time and place most convenient to the student. Students were asked what time of the day they viewed the videos. The question was a 'pick all that apply' question and thus the percentages do not reconcile. The results suggest that 90.4% of students viewed them outside office hours. Comparatively, 61.5% viewed the content during work hours. Students were also surveyed about where they watched videos. The majority, 96.1%, stated that they watched the videos at home. Videos were also frequently watched at university, 47.1%, while usage in other locations, such as public transport was quite small, 7.8%.

Written responses suggest that students appreciated the flexibility of the flipped learning approach.

i preferred it due to the flexibility of the unit only needing to be at the the university for two contact hours allowed for more time at home to complete homework,assignments etc, and gave spare time at the university itself.

I liked flipped learning. The flexibility and the total amount of time saved from watching the video lectures ultimately improved my overall performance in this unit.

I found the flexibility helped me fit ICT171 around my lifestyle.

There were, however, a small number of students that felt the lack of an allocated or scheduled lecture time, hindered their motivation and engagement in the unit. The following are comments from these students:

The fact that I don't make time for them or think they are as important as normal lectures.

Motivation to keep on top of the video lectures and readings, it can be quite easily to fall behind

Students predominantly watched the videos on their PC, 73.1%, or laptop, 75.0%. Smaller numbers of students used their tablet, 21.2%, or smartphone 11.5%. Some students appreciated the ability to integrate the flexible content into they daily schedule. The following are responses to the question, *"What was the worst part of flipped learning?"*:

The fact that I could watch it on the train on the way to my class and have everything fresh in my mind, as opposed to watching it right before my class, being up a bit earlier to get to my class. It made it more efficient because instead of waking up that hour earlier I could wake up and head to my class and watch it on the train on the way to the lecture allowing everything to be fresh in my mind.

Some of the videos were computer aided demonstrations, and some videos may not have played on all devices. This may have caused the usage of tablets and smartphones to be less than originally anticipated as indicated by the following comments:

video file formats had problems running in browser or on some smartphones. limited me to watching them only at home or uni.

I had some issues with the videos not playing on my iMac.

A minority of students also seemed to suffer from technical problems:

During the last few weeks I haven't had Internet at home so I haven't been able to watch some of them, but I could have put more effort into downloading them while I was at university.

The state of the Internet in Australia does create some difficulties for some people in accessing these videos. This is a much greater problem than where the student has a physical lecture that they can choose to attend.

Although a range of different student experiences, preferences and issues are evident, when students were asked to indicate their level of agreement with the statement *"I found the flexibility of flipped learning beneficial"*, there was strong agreement with an average score of 4.21/5. This suggests that on average, the flexibility of flipped videos is favourable to most students.

5.2 Face-to-Face Interaction

One of the fears with replacing face-to-face lectures with pre-recorded videos is a possible reduction in the interaction with staff and peers. In the ISEA flipped classroom, students had only 2 hours of face-to-face contact time per week. For reference, other similar first year units run with a 2 hour lecture and 2 hour tutorial. In response to the statement: *"Compared with traditional units, I interacted more with my peers in the flipped classroom"*, there was general agreement with an average score of 3.59/5. In response to the statement, *"Compared with traditional units, I interacted more with my instructors in the flipped classroom"* there was a similar level of agreement, 3.65/5. This suggests that, despite a reduction in overall class time, the interaction with peers and instructors was higher. This may have occured because of better quality interaction in the tutorial as the following comment indicates: *"The interactive tutorials were more engaging, interesting and I felt I learnt more from them than usual tutorials"*. It must however, also be acknowledged that the unit coordinators took responsibility for a lot of the teaching in this unit. It is possible that their enthusiasm for the flipped learning approach approach had an impact on these results.

5.3 Student Perceptions of Video Types

Students were surveyed about their preferences of video types. The talking head with PowerPoint slides (Figure 3) has the closest resemblance to a traditional lecture and was also the least desirable, with a score of 4.84/7. The chalk and talk, Figure 2, received a slightly higher score of 5.14/7. The most applied element of the course was the computer aided demonstrations, Figure 1, which received the highest student score of 6.08/7. On the same survey page, students were asked about how much they enjoyed traditional lectures with the physical presence of the lecturer. The response to this question was the most neutral with an average rating of 4.06/7. A summary table showing student preferences for different video types is shown in Table 1. In this unit, students least liked the approach that most closely resembled the traditional lecture and students were most satisfied with the applied instruction. Overall, student indicated that they preferred video lectures to traditional face to face lectures.

Written student feedback also suggested that students appreciated the ability to follow along with computer aided demonstrations. These computer aided demonstrations were the elements that would be most difficult to replicate in a traditional live lecture:

Table 1: Student preferences for video types

Question	1	2	3	4	5	6	7	Mean
On a scale of 1 (Did not enjoy at all) to 7 (Very much enjoyed), How much did you enjoy the blackboard style?	1	1	4	4	19	16	5	5.14
On a scale of 1 (Did not enjoy at all) to 7 (Very much enjoyed), How much did you enjoy the computer aided demonstrations?	0	0	2	1	7	21	19	6.08
On a scale of 1 (Did not enjoy at all) to 7 (Very much enjoyed), How much did you enjoy the talking head and PowerPoint style?	0	2	3	8	13	16	3	4.84
On a scale of 1 (Did not enjoy at all) to 7 (Very much enjoyed), How much do you enjoy traditional lectures, with the physical presence of the lecturer?	1	4	10	20	8	5	2	4.06

The ease at which the content was accessible and the ability to work along with the video. I found that it helped me discover and feel more engaged with my learning as opposed to sitting back and listening to a lecture.

i enjoyed the flipped learning system because it gave me the chance to watch the video lectures when i felt it was most appropriate time. It also gave me the chance to work along side on my pc as the video was playing

I find long lectures very boring and find it hard to concentrate. I do enjoy the video formats particularly as with this course they were broken down into specific topics. Which meant I could follow along and recreate on my own computer the various server things that we're being done.

Yes, I generally did as you can wind it back if you missed something.

Although many students, 34.7%, liked to see the face of the person speaking, the majority felt that it was not important. Given the ambivalence of the audience and the work involved, this is an aspect to give careful consideration when developing flipped learning videos. One possibility is the use of short video "bumpers". Bumpers are short video introductions and conclusions that are placed at either end of the video to present a human feel to the presentation with the remaining content consisting of a voice accompaniment to static PowerPoint slides or other visual elements. This approach might achieve a workable balance between some viewers' need to see the presenter's face and the developer's need to limit time spent on the editing process.

5.4 Coverage of material

Pre-recorded semi scripted video lectures are generally shorter than the typical lecture. The average length of video materials was 10 minutes. The total content in each of the teaching weeks averaged 42 minutes 28 seconds, substantially less than the 100 minutes that might be expected in a typical 2 hour lecture time slot. When asked *"Do you feel that less content was delivered under flipped learning?"* the overwhelming response was that the level of content was not compromised. The following student comments were typical:

No - I think there was the same amount of VALUABLE content delivered. This is my last unit, I wish this was in place for my whole degree.

I think it was much more direct, and kept my interest. However, I do think we miss out on the interesting tangents that occur in the lecture theatre.

Possibly less in terms of minutes, but more effective than a 2 hour lecture where you are losing concentration by covering too many topics at once.

No, too much time in lectures is spent arriving, quieting down the lecture theatre etc. I feel the weekly videos provide a more focused environment. I also felt the weekly videos were more rich in content and of a higher quality than the average lecture.

I don't think that less information was covered and that it was just covered in a more concise way with more direct information and less filling that can sometime happen in a defined 2 hr lecture time.

The downside to focused and edited content is that the presentations can become mechanical and less personal. Some students stated that they missed the tangents that lecturers sometimes go off on in face-to-face lectures.

5.5 General student perceptions

General student perceptions of flipped learning were positive. The mean and standard deviation of questions asked about general student perceptions are shown in Table 2. This was also evident in the written feedback from students:

I liked flipped learning in ICT171 because its a better learning experience than a traditional lecture/tutorial. Sometimes traditional lectures go on for so long and some information is lost. With flipped learning tutorial videos shows and give us a good understanding of what it is to do in the labs. Once in class we know exactly what we are doing and any questions can be answered by the tutors. For me Flipped learning is far better experiences learning and more units should implement this method of learning.

I enjoyed the short videos as I found it is much easier to concentrate and take in small videos than a 2 hr lecture. I found the videos to be very useful and related well to the information that was presented in the labs.

Students were surveyed on where they felt like they learnt the most. The majority, 61.5%, felt that they learnt the most in tutorials and 30.8% felt that they learnt the most in lectures. Only a minority of students, 3.8%, felt that they learnt the most in readings and assessment respectively. The transition from a traditional lecture-tutorial format to flipped learning forced considerable efforts and emphasis into the new

Table 2: Summary statistics on general student perceptions

Question	SD	D	N	A	SA	Mean
More university units should use flipped learning	0	3	8	34	7	3.87
Short flipped learning videos are more effective than traditional face-to-face lectures	1	3	8	26	13	3.92
My experience in ICT171 would have been better with at traditional lecture tute format	7	20	20	4	1	2.46

element, the flipped videos. While it is important that these video are well produced and accessible, the results of the survey reinforce the notion that the lectures are present to reinforce the tutorials. Reducing the emphasis on passive, one-way instruction methods is well supported by the literature (Hamdan 2013).

5.6 Future Work

The results described in this paper are preliminary and are purely based on the feedback of an electronic survey delivered to students. Consent was obtained from participants and permission was granted by the Murdoch Human Research Ethics Committee to utilise learning management statistics and data from student records. Future work will provide a more detailed and comprehensive analysis based on this data.

6 Conclusion

Students expressed a strong preference for the flipped learning model. Students liked the convenience and accessibility of the video lectures and in the IT university cohort, student preferences were to view the content outside of standard work hours. Although no group assessment was performed and contact hours were halved under flipped learning, students felt like they interacted more with their peers and instructors.

Students liked the concise nature of the video lectures and generally felt that an equivalent amount of content was covered in significantly less time. Three different video types were used and students expressed a preference for the computer led video where the applied elements of the unit were being demonstrated and discussed. The responses indicate that students often paused, rewound and followed along at home on their own PC. The unit coordinators found these video types the most straight forward from a production perspective. The students least liked the PowerPoint and talking head video type. It is possible that the reason for this was because the talking head video type was used to deliver the majority of the theory in the unit. Videos containing the instructor were the most difficult to produce due to the complexity of editing and many students reported that seeing the instructor's face was unimportant. The instructors found that, despite the emphasis on videos when converting to flipped learning, it is important to recognise that their role is to facilitate the tutorials, where the bulk of learning was reported to occur. Overall, the unit and the flipped learning approach was received very favourably by students.

References

Bishop J & Verlager MA 2013, 'The flipped classroom: A survey of the research', *120th Annual ASEE Annual Conference & Exposition Available*, Atlanta, USA, 23-26th June.

Butt A 2014, 'Student views on the use of a flipped classroom approach: Evidence from Australia', *Business Education and Accreditation*, 6(1) 33-43.

Davies RS, Dean DL & Ball N 2013, 'Flipping the classroom and instructional technology integration in a college-level information systems spreadsheet course', *Educational Technology Research and Development*, 61(4), 563-580.

Day JA & Foley JD 2006, 'Evaluating a web lecture intervention in a human-computer interaction course', *IEEE Transactions on Education*, 49(4), 420-431.

Freeman S, Eddy SL, McDonough M, Smith MK, Okoroafor N, Jordt H & Wenderoth MP 2014, 'Active learning increases student performance in science, engineering, and mathematics', *Proceedings of the National Academy of Sciences*, doi:10.1073/pnas.1319030111.

Gannod GC, Burge JE & Helmick MT 2008, 'Using the inverted classroom to teach software engineering', *Proceedings of the 30th international Conference on Software Engineering*, Leipzig, Germany, May, 777-786.

Hamdan N, McNight P, McNight K & Arfstrom KM 2013, 'A Review of Flipped Learning (white paper)', *Flipped Learning Network*, viewed August 2014, http://www.flippedlearning.org/.

Kong SC 2014, 'Developing information literacy and critical thinking skills through domain knowledge learning in digital classrooms: An experience of practicing flipped classroom strategy', *Computers & Education*, 78, 160-173.

Pierce R & Fox J 2012, 'Vodcasts and active-learning exercises in a "Flipped Classroom" model of a renal pharmacotherapy module', *American Journal of Pharmaceutical Education*, 76(10): 196, doi: 10.5688ajpe7610196

Teaching Computational Thinking in K-6: The CSER Digital Technologies MOOC

Katrina Falkner **Rebecca Vivian** **Nickolas Falkner**
School of Computer Science
The University of Adelaide
Adelaide, South Australia

`Firstname.lastname@adelaide.edu.au`

Abstract

In recent decades, ICT curriculum in K-10 has typically focussed on ICT as a tool, with the development of digital literacy being the key requirement. Areas such as computer science (CS) or computational thinking (CT) were typically isolated into senior secondary programs, with a focus on programming and algorithm development, when they were considered at all. New curricula introduced in England, and currently awaiting minister endorsement within Australia, have identified the need to educate for *both* digital literacy and CS, and the need to promote both from the commencement of schooling. This has presented significant challenges for teachers within this space, as they generally do not have the disciplinary knowledge to teach new computing curriculum and pedagogy in the early years is currently underdeveloped.

In this paper, we introduce the CSER Digital Technologies MOOC, assisting teachers in the development of the fundamental knowledge of CT and the Australian Digital Technologies curriculum component. We describe our course structure, and key mechanisms for building a learning community within a MOOC context. We identify key challenges that teachers have identified in mastering this new curriculum, highlighting areas of future research in the teaching and learning of CT in K-6.

Keywords: National curriculum, computer science, computational thinking, education, primary school, high school.

1 Introduction

Over the past decade ICT education has transitioned from focusing on ICT as a *tool* - with the development of digital literacy as the key requirement - toward understanding the underpinning concepts and workings of digital technologies. Areas such as Computer Science (CS) or computational thinking (CT) were typically isolated into senior secondary programs, with a focus on programming and algorithm development, when they were considered at all. The lack of computing curriculum at the primary level was perceived to be 'failing to provide students with access to the key academic discipline of CS, despite the fact that it is intimately linked with current concerns regarding national competitiveness' (Gal-Ezer and Stephenson, 2009).

To promote CS career pathways, global initiatives have targeted youth engagement and interest in CS through various outreach programs (Bell et al, 2011; Koppi et al., 2013; Lambert & Guiffre, 2009; Liu et al, 2011; Myketiak et al 2012). However, research findings and a continued lack of uptake of CS degrees suggest that outreach programs have had little success (Koppi et al., 2013). More recently, a drive to include computing in schooling curriculum has arisen, proposing that all children should have an opportunity to develop CT skills and have opportunities to be 'creators' of digital technologies (Gander et al., 2013; The Royal Society, 2012).

New curricula introduced in England (Department for Education, 2013), Australia (ACARA, 2012), New Zealand and the new ACM CS standards (Seehorn et al., 2011) have identified the need to educate for *both* digital literacy and CS, and the need to promote both learning areas from the commencement of schooling through to high school, to support the future generation of digital creators and increase international competitiveness. This is a significant milestone yet also raises a number of challenges, including the preparation of teachers and development of resources to support the success of implementation at a national scale. Curriculum change is not easy for teachers, in any context, and to ensure teachers are supported, scaled solutions are required. A potentially key factor in the success of implementing a new computing learning area will be appropriate professional development (PD) that provides teachers with the confidence and experience to integrate CS effectively into their classroom activities.

One educational approach that has gained traction for delivering content to large-scale audiences are massively open online courses (MOOCs), however, little is known about what constitutes effective MOOC design; particularly within the contexts of CT and teacher professional development. In this paper, we introduce the CSER Digital Technologies MOOC, assisting teachers in development fundamental knowledge of CT and the Australian Digital Technologies curriculum component. We describe our course structure, approaches to teaching CT within the K-6 context, and key mechanisms for building a learning community within a MOOC context. We identify key challenges that teachers have identified in mastering this new curriculum, highlighting areas of future research in the teaching and learning of CT.

2 The Australian National Curriculum

The Australian primary and secondary school system is undergoing a significant period of change, with the introduction of a National Curriculum. In Australia primary school includes the first year of school, called Foundation (F), also known as Kindergarten (K), until year 6 or 7, depending on the state. Secondary school (also known as high school) includes years 7 or 8 to year 12. The Australian Curriculum describes the nature of learners and curriculum across three broad year-groupings: Foundation to Year 2 (ages 5-7); Years 3 to 6 (ages 8-11); and Years 7 to 10 (ages 12-16).

In 2013, the Australian Curriculum Assessment and Reporting Authority (ACARA) released a series of draft curriculum standards for the national curriculum to be introduced across Australia in 2014. The curriculum introduces new learning areas with considerable effort committed in the definition of the curriculum and national achievement standards for each area. Some learning areas have achievement standards defined from K-12, while others, including ICT, have achievement standards defined from K-10, with decisions in the senior years of schooling to be defined at a later stage.

'The Shape of the Australian Curriculum' (ACARA, 2012), identifies that 'rapid and continuing advances in ICT are changing the ways people share, use, develop and process information and technology, and young people need to be highly skilled in ICT'. The ACARA documents include ICT awareness (digital literacy) as a key capability, embedded throughout the curriculum, and introduce a new learning area, Technologies, combining the 'distinct but related' areas of Design and Technologies and Digital Technologies (DT) (ACARA, 2013). DT focuses on developing knowledge of digital systems, information management and the CT required to create digital solutions. The core is the development of CT skills: problem solving strategies and techniques that assist in the design and use of algorithms and models.

The DT curriculum does involve some (CS) knowledge and skills, as well as some digital solutions (possibly involving programming and CS concepts) but the intended focus is on developing computational thinking, logic and problem solving capabilities. The DT curriculum is based on a *systems thinking* approach, designed to encourage students to understand the individual parts of the system, while also being capable of having a holistic view of the, including ethical, societal and sustainability considerations.

DT focuses on developing knowledge of digital systems, information management and the computational thinking required to create digital solutions. The core is the development of computational thinking skills: problem solving strategies and techniques that assist in the design and use of algorithms and models. The Australian Curriculum describes the nature of learners and curriculum across three broad year-groupings: Foundation to Year 2 (ages 5-7); Years 3 to 6 (ages 8-11); and Years 7 to 10 (ages 12-16).

Approaches to teaching vary according to the curriculum year-groupings. The development of both digital literacy and CT commences in the K-2 band and learning is based around directed play, facilitating students in developing an understanding of the relationship between the real and virtual worlds, the use and purpose of technology in communication, and the importance of precise instructions and simple problem solving in the digital world. In Years 3-6, students are guided to develop a wider understanding of the impact of technology, including family and community considerations, and are able to work on, and communicate about, more complex and elaborate problems. In this year level, students begin to apply CT to develop algorithms with visual programming software. Across Years 7-10, students move beyond their initial community and are required to consider broader ethical and societal considerations. In this band, students should be able to solve sophisticated problems using technology, and understand complex and abstract processes. Students begin to apply CT in their use of general-purpose programming languages to solve problems and create digital solutions. This development from K-10 supports the understanding of the utility of technology, as well as the development of problem solving skills and an abstract understanding of CS.

The eight key concepts that underpin the DT curriculum are allocated to one of two strands: 'Knowledge and Understanding' and 'Processes and Production Skills'.

2.1.1 Knowledge and Understanding

The Knowledge and Understanding strand builds awareness of digital systems and digital information. This includes the impact of digital technologies upon societies and relationships between these technologies and a society, exploring ethical and cultural considerations, from both a local and global perspective. The following sequence of learning objectives explores how an understanding of digital representation is developed across the curriculum:

- K-2: *Recognise and play with patterns in data and represent data as pictures, symbols and diagrams.*
- 3-6: *Explain how digital systems represent whole numbers as a basis for representing all types of data.*
- 7-10: *Explain how text, audio, image and video data are stored in binary with compression.*

2.1.2 Processes and Production Skills

In Processes and Production Skills, students explore how to solve computational problems, involving developing skills in 'formulating and investigating problems; analysing and creating digital solutions; representing and evaluating solutions; and utilising skills of creativity, innovation and enterprise for sustainable patterns of living' (ACARA, 2013).

The following presents an example sequence of learning objectives designed to introduce algorithmic planning:

- K-2: *Follow, describe, represent and play with a sequence of steps and decisions needed to solve simple problems*.
- 3-4: *Design and implement simple visual programs with user input and branching.*
- 5-6: *Follow, modify and describe simple algorithms, involving sequence of steps, decisions and repetitions that are represented diagrammatically and in plain English.*
- 7-8: *Develop and modify programs with user interfaces involving branching, repetition or iteration and subprograms in a general-purpose programming language.*
- 9-10: *Collaboratively develop modular digital solutions, applying appropriate algorithms and data structures using visual, object-oriented and/or scripting tools and environments.*

The processes and production strand encapsulates the key concepts of CT and presents challenges to us as a community in how we develop relevant skills within the younger age groups.

2.2 Challenges of New Computing Curriculum

The challenges faced by both nations in the adoption of these curricula are extensive. Consultation with Industry, Community and Education within Australia (ACARA, 2013b) has identified significant concerns in relation to teacher development (particularly at K-7), appropriate pedagogy, and skills needed for integration of DT learning objectives with the teaching of other learning areas. Respondents (55%) indicated concern with the manageability of the implementation of the DT curriculum and 45% of respondents did not think that the learning objectives were realistic. Further concerns were expressed regarding teacher preparation.

Bell et al (2012) describe the New Zealand experience of the rapid introduction of a senior secondary CS curriculum, and the need for extensive teacher development that addresses both content knowledge and pedagogical knowledge. In the Australian curriculum, this will involve teachers understanding CT, CS disciplinary knowledge as well as the development of skills in visual or general-purpose programming. Further, it has been recommended that key to teacher development will be the integration of aspects such as CT across other learning areas (Yadav et al 2011). In their CT course for educators, the instructors recommend incorporating CT modules into teacher education courses to expose teachers to these ideas. Through connecting CT to learning areas, it is recommended that teachers will be able to move beyond an 'abstract' idea of CT and understand its application and relevance as a problem-solving tool. Ragonis, Hazzan, and Gal-Ezer (2010) identify best practice as the development of a dedicated teacher development programme specifically addressing CS. They recommend that a critical element of such programs is to use empirical research to guide appropriate pedagogy for specific year bands, and learning objectives.

However, despite materials being available to teachers through PD, Settle et al. (2012), recognise the difficulty teachers face in translating materials into existing curriculum, when unfamiliar with the tools. In a study by Meerbaum-Salant et al (2011), they identified that even teachers experienced in CS, can be challenges with the introduction of new tools, which created feelings of anxiety, and resulted in teachers to deviate from original lesson plans. Another issue regarding tools is that they may be suggested to teachers to use to teach subject matter but they may not always be available. Tinapple, Sadauskas, and Olson (2013) further comment on the challenge for teachers, where expected software and/or hardware are not easily available. This is a consideration that needs to be taken into account with national computing PD, particularly when teachers from a variety of contexts (e.g. rural, disadvantaged) may be participating. Such findings indicate that teachers require opportunities to explore tools and also alternative 'unplugged' lessons as well as a variety of potential software that could be used.

In our previous review of research in the teaching and learning of CT within K-12 (Falkner, Vivian & Falkner, 2014), we identified a dearth of research into the development of appropriate pedagogy within the K-10 space, and in particular, within the K-6 space, with most of the research that has been done is situated within outreach programs, focussed on sharing teaching techniques aimed at motivating students to study CS, to address negative perceptions of the discipline, stereotypes and to increase diversity in our student cohorts. This places extreme pressure on deliverers of PD as well as teachers, when pedagogy and pedagogical strategies are underdeveloped in the K-10 space of computing education.

Developing pedagogically appropriate lessons for particular contexts, needs and students may be challenging for teachers and the adoption of teaching approaches may be influences by teacher confidence in teaching the learning area. In one study, when teachers used guiding activity resources for their CS lessons, they were apprehensive about using teaching methods such as group work (Curzon et al, 2009). Further, teachers felt that because they were unfamiliar with the topic, considerable preparation would be required. In Black *et al*'s survey (2013), they discovered that teachers tend to focus more on fun activities rather than providing opportunities for deep learning of CT, focussing on impressive technology, physical computing and programming in constructionist environments. These forms of activities can complicate the learning environment further by placing additional stress on teachers inexperienced with technology.

Support for the professional development of teachers is crucial in expanding CS curricula, including the creation of community networks to share insights and pedagogical approaches and research (ACARA, 2013; Gander et al, 2012). This was confirmed by a study by Black et al. (2013) involving a survey of UK computing teachers in relation to their suggestions on improving CS education, and teacher development needs. Although their results highlighted a need for teacher training, they also expressed the need for a network and community to support resource development.

3 Massive Open Online Courses

Massively open online courses (MOOCs) offer one means to deliver education at scale and have the potential for community elements to connect participants across various locations, even around the world. Although online learning is not new, it has been argued that the difference between online learning and MOOC environments are the combination of teaching approaches course instructors use, the massive levels of participation and the openness (Glance, Forsey, & Riley, 2013).

Typically two different types of MOOCs have been identified; one being based on courses that embrace the use of videos to deliver content and computer-assisted online assessment ('xMOOCs') (Glance et al., 2013) and other courses based around online communities and connectivist principles, called 'cMOOCs' (Siemens, 2005, 2012). A number of 'hybrid' MOOC versions are also surfacing, that combine a mixture of xMOOC and cMOOC approaches, blending a structured pace with a focus on participant-led communities, such as EDMOOC by Coursera, and MOOC-EDs introduced by the Friday Institute (Kleinman, Wolf, & Frye, 2013).

Enrolment in MOOCs have reported significantly high enrolment rates, with edX and MITx reporting a total of 841,687 registrations from the fall of 2012 to the summer of 2013 across a number of their courses (Ho et al., 2014). In that year, 43,196 participants earned completion certificates. On average there was a 5.17% completion rate across the courses, with a 9% completion rate for those who went beyond 'enrolment' in the course. A typical measure of completion within xMOOCs is the completion rate for those that complete half or more of the course, known as *explorers* – edX and MITx report a completion rate for explorers of 54%. A supporting component of xMOOCs are the community forums, which have seen engagement anywhere from 6.5% to 25.7% with an average of 7.9%.

In comparison, cMOOCs measure enrolment based on members who 'subscribe' to the course via mailing lists or by signing up to the course platform. cMOOC enrolment figures have been found to be ranging from the hundreds to the low- thousands and researchers typically report participant engagement through the measurement of social media activity (de Waard et al., 2011). While the communities engagement seems large and broad, analysis of cMOOC social media engagement reveals that typically a small core of participants generate the activity. For example, in CCK11, 18% (N= 126) participants were actively involved (Kop, Fournier, & Mak, 2011) and in First Steps in Teaching and Learning (FSTL12) (Roberts, 2012) about 30% actively participated throughout the 6 weeks and only 14 participants undertook assessment and received a certificate.

4 The CSER Digital Technologies MOOC

In selecting a 'hybrid' MOOC approach, we were able to deliver structured content as well as adopt a participant-led community, which is proposed as being valuable for teacher support in computing curriculum implementation (ACARA, 2013; Gander et al, 2012; Black et al. 2013). A large focus of the Australian DT curriculum is on CT, which is defined in the ACARA curriculum documents, as '*a problem-solving method that involves various techniques and strategies, such as organising data logically, breaking down problems into components, and the design and use of algorithms, patterns and models*'. Understanding CT involves understanding core CS concepts, and the ability to conceptualise and create abstractions that define solutions to problems.

At the level of K-6, the teaching and learning of CT involves the developing of capabilities in solving problems, utilising core concepts such as algorithm definition – including the introduction of selection and iteration – and data collection and analysis. Also introduced are key ideas such as abstraction and decomposition. Previous work in educator PD recommends integrating new concepts throughout courses and the application of concepts to other learning areas (Yadav et al. 2011). CT concepts and ideas were presented throughout the MOOC modules and examples of the concepts (e.g. abstraction, decomposition) were defined, incorporating lesson ideas with application to everyday examples and other learning areas.

4.1 Course context

The average age of primary teachers is 42.1 and 44.5 for high school teachers, with leadership roles being held by those around 50 years of age (Cordova, Eaton, & Taylor, 2011). In Australia, the teacher workforce is predominately female, particularly in the primary years (81% of primary teachers and 57% for secondary teachers). In Australia, teachers are reportedly spending 46 hours per week on all school related activities and about 8 or 9 days a year toward professional learning (Cordova et al., 2011).

Australian primary school teachers are typically generalist teachers, with 80% reportedly teaching in generalist classrooms (Cordova et al., 2011), implementing the various learning areas prescribed by their state or territory. Some schools are fortunate enough to have specialist teachers, such as an ICT teacher, but this is not typically the case for all schools, with only 6% (N=7,500) of teachers reportedly teaching computing (Cordova et al., 2011). In Australia, 17% of teachers report having had some post-secondary education in computing, with only 8% having been trained in the practice and pedagogy of computing (Cordova et al., 2011). Teachers are typically left to integrate the use of ICTs and digital literacy into their classroom activities by integrating with other learning areas.

4.2 Course Structure and Design

In response to existing research findings, we identified in the development of this course the importance of providing learning and teaching opportunities that were tool-independent and focussed on deep learning (Black et al, 2013, Meerbum-Salant et al, 2011), and the need to provide exemplars of activities that were already integrated with existing knowledge areas within the curriculum – removing the need for direct translation (Settle et al, 2012). We drew on and adapted existing lesson ideas from organisations and initiatives such as CS Unplugged and Code.org, and drew on lesson ideas and approaches from education texts in other learning areas, such as Mathematics, Science and Literacy, and with examples from possible teaching themes, commonly used within K-6.

As a new learning area, the disciplinary content would be new for many teachers. Therefore, the course was designed around a series of seven topics that align with the Australian curriculum, delivered in a logical order, suitable for someone learning CS for the first time over a period. Our goal in the first unit was to provide an introduction to digital technologies, showcasing the development and application of digital solutions to solve real-world problems. Further, we wanted to define terminology for digital technologies (e.g. computing and CS) and distinguish between *digital literacy and digital technology creation and CT*. In unit 2, the more familiar topics of patterns (creating and continuing sequences and recognition) and data representation (collecting and representing data in different ways, with and without technology) were introduced because of the potential links to what teachers are already doing in Mathematics and Science. In subsequent units, we moved toward the use and application of data by computers and digital data as well as the introduction of more abstract concepts, such as algorithms. Lastly, we visited visual programming environments.

In each unit, the topic (e.g. 'digital systems') was introduced with the relevant Australian learning objectives. Each unit were broken into sub-topics and for each sub-topic a concept video was created or an existing suitable video used in which the concept was explained and supported with analogies and real-world examples. Links were made to the Australian curriculum 'expected outcomes' as guiding points for assessment. The goal of the course was to deliver core disciplinary knowledge, packaged for primary year levels, and lesson ideas so that teachers could feel comfortable and empowered to create or draw on existing resources to design learning activities to meet the learning objectives. The sequence of units for the Digital Technologies course are outlined in Table 1.

Table 1: Sequence of MOOC Modules

Unit 1: Introduction
Unit 2: Data – Patterns & Play
Unit 3: Data - Representation
Unit 4: Digital Systems
Unit 5: Information Systems
Unit 6: Algorithms & Programming
Unit 7: Visual Programming

The core elements of the course were the focus on worked examples, and the sharing of further examples as identified and developed by the teachers themselves. To assist in community development, we initiated a Google+ community for the course; this space allowed participants to network, share ideas and course tasks and to collectively build an online series of resources corresponding to topic areas.

Each unit incorporated two fully worked examples of how specific learning objectives could be addressed across K-6. For example, Unit 6: Algorithms & Programming, incorporated a worked example exploring instructions and sequences of instructions addressing the learning objectives (with ACARA id):

- Follow, describe and represent a sequence of steps and decisions (algorithms) needed to solve simple problems (ACTDIP004)
- Define simple problems, and describe and follow a sequence of steps and decisions (algorithms) needed to solve them (ACTDIP010).

Within this worked example, we explore multiple learning and teaching activities for different age groups connected to different knowledge areas, starting with a wriggle break activity, commonly included to signpost a change between activities in early years. A variety of instructions are written on pop sticks, such as "jump up and down", or "spin around". A student or the teacher selects a pop stick at random, and the class acts out the selected activity, with opportunities for paired or group exploration to demonstrate achievement. This activity is deceptively simple – while fun, and engaging for the students, it introduces the idea of instructions, and sequences of instructions through variants of the game. We then explore an extension of this activity, designed to assist in literacy development, where students construct a sentence, word by word, through rearranging a series of pop sticks (see Figure 1).

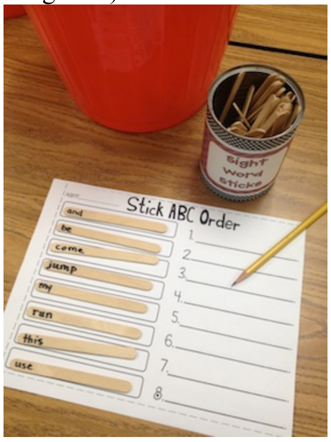


Figure 1: Computational Thinking development embedded within Literacy (Tunstall, 2013).

We continue this worked example, by exploring learning and teaching activities where students are able to construct their own instructions, incorporating existing videos and examples, including the well known "jam sandwich" example (Bagge 2012), and identify ideas for assessment of these learning objectives:

- Students understand that computers require explicit instructions.
- Students can explain that an algorithm is a step-by-step sequence of instructions.
- Students can re-order instructions or develop instructions that form a logical sequence.
- Students can adapt instructions based on their observation of an outcome.
- Students use descriptive and precise language when giving instructions.
- Students can provide a set of instructions to achieve a desired outcome.

For each unit, teachers were asked to post a task on the Google+ community page for the course. These tasks were designed to be informal and promote the exchange of tools, resources and lesson ideas. In all cases, teachers were provided with three options so that we could have a variation of content being shared. For example, within the same unit, we considered: '*Design an activity that explores sequences of instructions'*.

As an indication of the type of engagement within the course, we include here two example activities shared by teachers within the course. Teacher A integrated a lesson within the context of Mathematics, incorporating Beebots as an interactive technology at Year 1; the activity required students to identify the sequence of instructions to navigate their Beebot between two points on a map. This activity required the students to develop and demonstrate key skills in problem solving, instruction selection, sequencing and navigational language.

Teacher B identified a lesson activity that could be either represented on its own, or within the Drama learning area in the Arts curriculum at a more senior year level (Years 3 or 4), where students were asked to define a sequence of instructions for a random scenario, such as brushing teeth, or packing a school bag, which were then enacted by the group in a performance. This activity required the students to develop and demonstrate key skills in problem solving, instruction design and selection, sequencing, selection and repetition. Further integrating aspects of digital systems, they suggested a simplification of the activity suitable for Year 1, adopting QR codes as a mechanism for accessing a set of existing instructions for a given scenario.

Teachers were able to comment on peers' task contributions in the Google+ community by providing constructive advice or suggesting extensions or ways that the activity could be adapted for other learning contexts.

Of the 1374 people who enrolled in the course via the course website, we had 473 participants connect to the Google+ community. Counting Google+ community posts of the community revealed that posting activities began at 269 for unit 1 and tapered off to around 100 in for the portfolio.

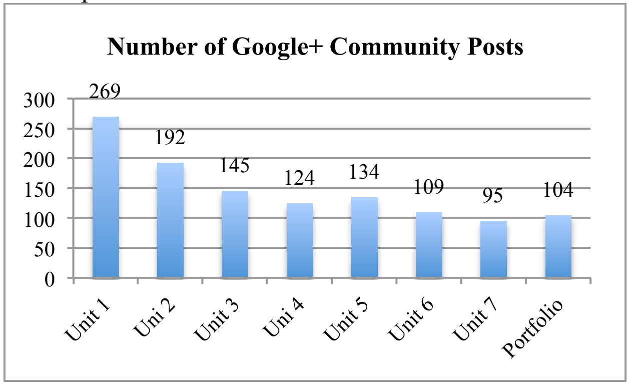


Figure 2: Google+ Community Posts

While we felt a level of enthusiasm from the community and participants appeared to be actively posting, what was participation and engagement like across the course and how did teachers experience learning digital technologies content and partaking in the course? The follow section reports on data obtained about participant engagement and survey findings about participant experiences.

5 Cohort Participation

Although we did not originally ask participants for demographic details, we were able to gather some idea of where participants were located via an anonymous survey as part of an optional activity in unit 3. The majority of participants appear to be from South Australia, Queensland, New South Wales and Victoria (see Figure 3, N=174). Advertising and visits generally covered these areas, suggesting that for future courses, more targeted advertisements and connections need to be made to Western Australia, Northern Territory and Tasmania.

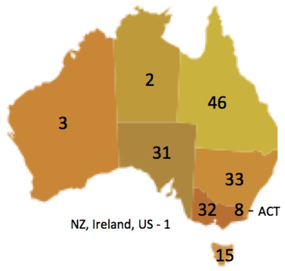


Figure 3: Survey results for location of participants

Unsurprisingly, with the majority of teachers being female in Australia (81%) and the average age of teachers being between 40 and 50, according to the YouTube analytics, the majority of the cohort was female and between the age bracket of 45 to 64 (see Figure 4). These results show that we were able to target our intended audience and attract a female demographic that is typically lacking in post-secondary courses and careers (Koppi et al., 2013).

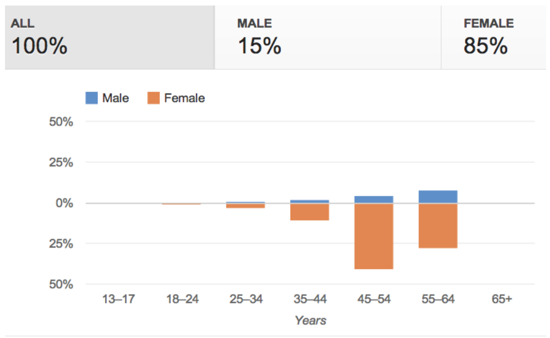


Figure 4: Google Analytics demographic details for video views

Of the 1378 enrolled in the course, 99 participants completed the course and 438 did not engage in the course any further than enrolling. As a result, we have a 7.2% completion rate, or 10.5% completion rate for those who went ahead and began the course. When considering completion rates, and measures of MOOC engagement, we consider engagement across all course components, and within core components specifically. Our completion rate overall was 7.2%, with a further 5.73% of participants exploring half or more of the course (without completion), and 56.39% of the participants completing less than half of the course. In terms of core components only, 8.13% of the participants explored half or more of the core components (without completion) and 52.3% of the cohort (group of MOOC participants) explored less than 50% of the core components. Our completion rate for explorers was 55.7%, and 46.9% when considering core components only.

Overall, across the course platform and the Google+ community, the completion rates were mostly in-line with what one would expect to see in MOOCs in terms of enrolment and completion. However, 34.3% of the cohort (n=473) viewed and/or engaged with the online community – a significant increase in engagement over typical MOOCs. The completion rate relative to those that engaged with the community is 20.9%. A key motivating factor for this engagement was tying the course tasks in with the use of the Google+ community – a strategy that resulted in the co-creation of K-6 digital technologies resources and lesson plans.

In accordance with the participation and engagement described previously, we had a high number of viewers watching videos during the first unit (N=462), slowly decreasing during each module (to 66 in unit 7). According to the YouTube analytics, the average video length created by the CS Education Research group, was 5.8 minutes – ranging from around 1 minute to 11 minutes. This timeframe is typical of many of the xMOOC style video length. The average length of the videos watched was 4.37 minutes, suggesting that short concept videos work; however, designers need to consider presenting important information at the beginning.

5.1 Survey Responses

5.1.1 Survey: Participant Background

Some 51 participants responded to a survey about their experience in the MOOC. 45 females and 5 males responded. We recognize that the sample is biased towards educators who completed the course but the participants are able to provide insight into their experience across the whole course. Almost 50% (23%) of survey participants said that they had not participated in an online course before (28 had). A majority of the survey participants were teachers in primary schools, with some people in leadership roles or enrolled in university pre-services courses (see Table 2).

Table 2: Survey participants' professional role

Professional Role	Count
Teacher in Primary School	28
Teacher in multi-level school (K-12)	6
Leadership Role in an ICT-related area	5
Student/Pre-service teacher	4
Teacher in High School	2
IT-related professional	1
Leadership role (in other area or organisation)	1
Missing	4
Grand Total	51

We received survey responses from most people throughout Australia, except from the Northern Territory and Western Australia. Those who did respond were from Queensland (16), Victoria (11), New South Wales (10), South Australia (7), Tasmania (4), Australian Capital Territory (2) and one from the United Kingdom.

The MOOC was targeted at the Foundation to Year 6 levels, however, we had a number of people enroll who were not specifically focused on these years in their professional roles. Table 4 shows that survey participants were primarily from the K-6 years but also worked across multiple year levels or upper year levels.

Table 3: Survey Participants' Year Level

Year Levels	Count
K-6 Years	19
Years 7-12	8
Multiple	16
Higher Ed/ Other	2
Not a teacher	3
Missing	3
Total	51

Table 3 demonstrates that survey respondents were from a variety of different Year levels, with most from either the K-6 years or working across multiple years.

Table 4 represents participant confidence (on a scale of 0 to 7, with 0 being no experience at all and 7 being highly confident), previous experience in teaching digital technologies, using ICT in everyday activities and confidence implementing digital technologies learning activities (N=51).

Table 4: Survey participants' confidence before the MOOC

	Experience Teaching DT	Using ICT course	Implementing DT
Mean	3.9	6.2	3.5
Median	4	6	4
Mode	4	7	4
Std. Dev	2.143	0.967	1.88
Min	0	2	0
Max	7	7	7

Before going into the course, participants reported that they were reasonably comfortable using ICTs in their everyday lives and activities (mean 6.2) but had limited experience teaching digital technologies (mean 3.9) and confidence implementing the learning area (mean 3.5).

Table 5: Survey participants' previous experience with DT activities

Previous experience with digital technologies	Count
No previous experience	22
Visual Programming	12
Programming (general-purpose)	8
Other (basic computer use, internet, etc)	6
Microworlds LOGO	1
Robotics	1
Algorithm activities	1
3D simulations	1

Table 5 presents participant experience with digital technologies lessons before starting the course. Many had no previous experience (22 responses), with some having had experience in using visual programming (12), general-purpose programming (8) and teaching students about basic computer, Internet and application use (6). Other items that mentioned were algorithm activities, 3D simulations, robotics and LOGO.

5.1.2 Participant experience in the course

A majority of participants reported completing all of the modules, which is consistent with our participation findings that those who completed the portfolio also completed all core and non-core modules. When survey participants were asked which modules they did not complete (6 in total), the following modules were identified in Table 6. Although some had not completed these modules they had intentions to return to view materials or, even if they did completed all modules, some still mentioned that they would revisit the modules in more detail. Others who had not tried visual programming during the course had set personal goals to learn visual programming in the near future.

Table 6: Survey participants' 'non completed' modules

Module	Count
Data - Representation	1
Digital Systems	3
Information Systems	2
Algorithms & Programming	1
Visual Programming	2
Portfolio	1

After the course, participant confidence to implement the new learning area had risen (mean 6.2) as well as confidence in implementing digital technologies lessons (mean 6.5) and making cross-curricula links (mean 6.2). However, we still have room to improve on increasing teacher confidence with digital technologies teaching strategies, the organization of content, designing activities, the integration of ICT and applying content knowledge. Survey participants were asked to select from a list of items, which activities they had tried for the first time since undertaking the course. Table 8 presents a summary of the frequency of participants who reported trying the new activities.

Table 7: Survey participants' confidence after the MOOC

Statement	**Mean**	**Mo de**	**Std. Dev**	**Min**	**Max**
Comfort Implementing DT AFTER	6.2	7	1.1	1	7
Confidence Implementing Lessons	6.5	7	0.7	5	7
Applying Content Knowledge	4.4	5	0.7	2	5
Organising Content	4.4	5	0.7	2	5
Designing Activities for Implementation	4.3	5	0.7	2	5
Teaching Strategies	4.2	5	0.8	2	5
Integrate ICT into lessons	4.5	5	0.6	2	5
Cross Curricula Links	6.2	7	1.0	2	7

Table 8: Survey participants' experience in DT activities after the MOOC

New activities tried	**Count**
Algorithm activities	24
Binary activities	23
Visual Programming	21
Other module topics	16
Data collection and analysis	13
Robotics	6
Code club	1

Previously 22 participants reported having had no experience with digital technologies activities or any of the items, but since participating in the course many had tried new activities for the first time. Many of the activities adopted for the first time were related to algorithms (24 responses), binary (23), visual programming (21), other module topics (16; digital systems, information systems) and data (13). Six participants had tried robotics and 1 person reported starting a school Code Club.

5.1.3 Perceived Challenges

Content analysis was applied to the participant responses to the questions about the challenges encountered by participating in the MOOC and the most challenging Modules. Time was a major factor identified as external pressures from work or personal life meant that they could not spend as much time on each module and activity as they would have liked.

Participants were asked to identify the most challenging modules. Participants reported that the later units from module 5 onwards (4 responses), algorithms and programming (9), binary (5) and other module 3 topics (3), and preparing the portfolio (3) were the most challenging. Two participants also mentioned that 'everything' was challenging. Using content analysis, we were able to group the reasons as to why participants felt that these topics or the MOOC in general was challenging. We identified a series of primary reasons that emerged relating to the *content being challenging*, *transferring and applying knowledge, personal challenges* and *external factors.*

The content was challenging: These reasons included that the topic was dry itself, that sometimes the information was overwhelming or too technical, or that they wanted to know more but were limited in time. A number of participants mentioned that the content was challenging because it was new (10 responses). Four participants mentioned that although the content was challenging it was exciting to learn and three mentioned that they intend to explore the course content further. Some interesting comments emerged around the language of the new curriculum – that once concepts, such as algorithms or iteration, were de-mystified they were far more comfortable with the new curriculum and that the terminology was less 'scary'.

Transferring and applying knowledge: This topic included reasons such as *that designing lessons for this new learning area was challenging* (4 responses) and *that transferring knowledge to the classroom* (3) or *the design of learning activities* (4) for the community was a challenge. One other participant who was in a leadership role mentioned that transferring knowledge in the MOOC for teachers would be a challenge for them because they needed to understand it well enough themselves first at a higher level to be able to transfer the knowledge.

Personal challenges: One participant expressed a feeling of pressure to perform well in the community by producing quality materials or lessons and another participant expressed that they personally felt that they struggled with a topic because they were not good at mathematics. Although these only account for 2 out of 51 responses, others in the community may have also encountered similar challenges.

Two *external challenges* were identified relating to *time constraints* (32 responses) such as personal reasons, workload, work pressures, life events that limited their ability to participate or complete modules as well as *technical challenges* (8), such as low internet speed, computer issues and limitations imposed by school contexts to access particular sites.

6 Conclusions

The expected changes in the teaching of CS represent a significant challenge for our schooling systems. CT and CS will form part of the Australian standard curriculum from K-12 from 2014. In this paper, we have described the CSER Digital Technologies MOOC, which supports K-6 teachers in their development of CT awareness, within the direct context of their learning and teaching activities. We have described our course structure, incorporating a specific example of how we have focussed course activities within the teacher's context, incorporating a range of learning examples, with varied tool dependency and integration across multiple existing knowledge areas.

Our analysis has indicated that this course can assist teachers in developing their understanding and confidence in CT and digital technologies. Similar to previous work that found weaving CT concepts throughout teacher courses was beneficial (Yadav et al 2011), we also found that unpacking core CT concepts (e.g. abstraction, algorithms, decomposition) and programming statements (e.g. functions, iteration) that featured in the curriculum, with everyday examples and cross-curriculum connections assisted teachers to understand and feel more comfortable the new curriculum. However, our cohort still indicated that the challenge of new content, and translation requirements for their immediate teaching context were still of concern, which is consistent with literature in the area. While we have provided one resource that addresses the required development of CT awareness, there is still substantial effort required not in providing needed resources, but also in further exploring appropriate pedagogy within the K-6 context. We identify that further research and development is required in building teaching strategies through exploring pedagogical research in K-6 digital technologies education and translating effective pedagogy to teachers through worked examples (e.g. pair programming, teamwork, problem-based learning, etc). Further, following teachers into the classroom to determine impact of such PD courses in this field is important.

Findings from the literature state that teachers suggest computing education PD incorporate online community networks to support teachers and facilitate the sharing of resources (ACARA, 2013b; Black et al, 2013; Gander et al, 2012;). A core, and tentatively successful, aspect of our course featured the development of a knowledge sharing community; our future work seeks to evaluate the community component and the more immediate and long-term impact use of the community had on teacher support and implementation.

7 References

ACARA. (2012): The shape of the Australian curriculum: technologies. Sydney, NSW: ACARA, http://www.acara.edu.au/curriculum_1/learning_areas/technologies.html, Accessed 17 Aug 2014.

ACARA. (2013): The Australian curriculum: technologies information sheet. Sydney, NSW: ACARA, http://www.acara.edu.au/curriculum_1/learning_areas/technologies.html, Accessed 17 Aug 2014.

Bagge, P (2012) Jam Sandwich Algorithm (programming teacher bot), Computing At School (CAS), University of Kent, available online, http://community.computingatschool.org.uk/resources/376

Bell, T., Curzon, P., Cutts, Q., Dagiene, V. & Haberman, B. (2011) 'Introducing students to computer science with programmes that don't emphasise programming', *Joint conference on Innovation and technology in computer science education*, Darmstadt, Germany.

Bell, T., Newton, H., Andreae, P., & Robins, A. (2012): The introduction of computer science to NZ high schools: an analysis of student work. *Workshop in Primary and Secondary Computing Education*, Hamburg, Germany, 5-15.

Black, J., Brodie, J., Curzon, P., Myketiak, C., McOwan, P., & Meagher, L. (2013): Making computing interesting to school students: teachers' perspectives. *Proc. ITiCSE*, Canterbury, England, 255-260.

Cordova, J., Eaton, V. & Taylor, K. (2011) 'Experiences in computer science wonderland: a success story with Alice', *Journal of Computing Sciences in Colleges,* vol. 26, no. 5, 16- 22.

Curzon, P., McOwan, P., Cutts, Q., & Bell, T. (2009): Enthusing & inspiring with reusable kinaesthetic activities. *SIGCSE Bulletin* **41**(3): 94- 98.

de Waard, I., Koutropoulos, A., Özdamar Keskin, N., Abajian, S., Hogue, R., Rodriguez, C., et al. (2011) 'Exploring the MOOC format as a pedagogical approach for mLearning', *10th World Conference on Mobile and Contextual Learning*, Beijing, China.

Department for Education. (2013): The national curriculum in England. Cheshire, UK: Crown.

Falkner, K, Vivian, R, & Falkner, N. (2014) The Australian Digital Technologies Curriculum: Challenge and Opportunity. In Proceedings of ACE 2014.

Gal-Ezer, J., & Stephenson, C. (2009): The current state of computer science in US high schools: a report from two national surveys. *Journal for Computing Teachers*, Spring: 1- 5.

Gander, W., Petit, A., Berry, G., Demo, B., Vahrenhold, J., McGettrick, A., Boyle, R., Drechsler, M., Mendelson, A., Stephenson, C., Ghezzi, C. & Meyer, B. (2013): Informatics education: Europe cannot afford to miss the boat ACM Europe: Informatics Education Report. New York.

Glance, D., Forsey, M. & Riley, M. (2013) 'The pedagogical foundations of massive open online courses', *First Monday*, vol. 18, no. 5- 6.

Ho, A., Reich, J., Nesterko, S., Seaton, D., Mullaney, T., Waldo, J., et al. (2014) HarvardX and MITx: The first year of open online courses, fall 2012- summer 2013, *HarvardX and MITx: The first year of open online courses (HarvardX and MITx Working Paper No. 1)*, Social Science Research Network: Social Science Electronic Publishing, [online] Available at: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2381263

Kleinman, G., Wolf, M. & Frye, D. (2013) *The digital learning transition MOOC for educators: exploring a scalable approach to professional development*, [online] Available at: http://www.mooc-ed.org/wp-content/uploads/2013/09/MOOC-Ed-1.pdf

Kop, R., Fournier, H. & Mak, J. (2011) 'A pedagogy of abundance or a pedagogy to support human beings? Participant support on Massive Open Online Courses', *International Review of Research in Open and Distance Learning,* vol. 12, no. 7, pp. 74- 93.

Koppi, T., Ogunbona, P., Armarego, J., Bailes, P., Hyland, P., McGill, T., et al. (2013) *Addressing ICT curriculum recommendations from surveys of academics, workplace graduates and employers*, [online] Available at: http://www.arneia.edu.au/project/37

Lambert, L. & Guiffre, H. (2009) 'Computer science outreach in an elementary school', *Journal of Compter Science in Colleges,* vol. 24, no. 3, pp. 118- 124.

Liu, J., Hasson, E., Barnett, Z. & Zhang, P. (2011) 'A survey on computer science K-12 outreach: teacher training programs', *Frontiers in Education Conference*, Rapid City, San Diego.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2010): Learning computer science concepts with scratch. *Proc. International workshop on computing education research*, Denmark, 69- 76.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011): Habits of programming in scratch. *Proc. ITiCSE,* Germany, 168- 172.

Myketiak, C., Curzon, P., Black, J., McOwan, P. & Meagher, L. (2012) 'cs4fn: a flexible model for computer science outreach', *Innovation and technology in computer science education*, pp. 297- 302, ACM, Haifa, Israel.

Ragonis, N., Hazzan, O., & Gal-Ezer, J. (2010): A survey of computer science teacher preparation programs in Israel tells us: computer science deserves a designated high school teacher preparation! *Proc. ACM technical symposium on computer science education,* 401- 405.

Roberts, G. (2012) *OpenLine project: final report*, Oxford Brookes University, [online] Available at: http://openbrookes.net/firststeps12/files/2012/08/OpenLinefinalreport2012-08-10_merged.pdf

Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., . . . Verno, A. (2011): CSTA K-12 computer science standards The CSTA Standards Task Force. New York: Computer Science Teachers Association, Association for Computing Machinery.

Settle, A., Franke, B., Hansen, R., Spaltro, F., Jurisson, C., Rennert-May, C., & Wildeman, B. (2012): Infusing computational thinking into the middle-and high-school curriculum. *Proc. ITiCSE*, Haifa, Israel, 22- 27.

Siemens, G. (2012), 'MOOCs are really a platform' *ELearnspace*, [online] Available at: http://www.elearnspace.org/blog/2012/07/25/moocs-are-really-a-platform/

The Royal Society. (2012): Shut down or restart? The way forward for computing in UK schools. London.

Tinapple, D., Sadauskas, J., & Olson, L. (2013): Digital culture creative classrooms (DC3): teaching 21st century proficiencies in high schools by engaging students in creative digital projects. *Proc. International Conference on Interaction Design and Children*, New York, 380- 383.

Tunstall, R (2013), Tunstall's Teaching Tidbits, available online, http://tunstalltimes.blogspot.com.au/2013/05/five-for-friday-holla.html

Yadav, A., Zhou, N., Mayfield, C., Hambrusch, S., & Korb, J. T. (2011). Introducing computational thinking in education courses. In *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11* (p. 465). New York, New York, USA: ACM Press. doi:10.1145/1953163.1953297

Why Don't More ICT Students Do PhDs?

Cally Guerin[1]*, Asangi Jayatilaka[2], Paul Calder[3], Alistair McCulloch[4], Damith Ranasinghe[2]

[1] School of Education / University of Adelaide, Adelaide, Australia
[2] School of Computer Science / University of Adelaide, Adelaide, Australia
[3] School of Computer Science, Engineering, and Mathematics / Flinders University, Adelaide, Australia
[4] Research Education (Learning and Teaching Unit) / University of South Australia, Adelaide, Australia

* cally.guerin@adelaide.edu.au

Abstract
Compared to many other disciplines, ICT has relatively few students choosing to continue into doctoral studies. We have explored some of the perceived barriers to undertaking doctoral studies in ICT in three Australian universities. Current students were surveyed to establish their post-course intentions regarding employment and further study. Their reasons for not choosing to go onto research degrees were linked largely to concerns about the financial implications of such study and a limited understanding of what research in ICT involves. We recommend that ICT students be given accurate information about the costs involved, that students have authentic undergraduate experiences of research, and that smooth pathways be developed to allow students to return to doctoral studies after working in industry.

Keywords: Information and Communication Technology (ICT); doctoral education; motivations; barriers.

1 Introduction
Despite increasing focus on doctoral-level education and the doctoral graduates produced by higher education institutions, relatively few ICT bachelor graduates from Australian universities choose to undertake doctoral studies compared to most other STEM disciplines (Graduate Careers Australia 2013). This paper seeks to uncover and explore some of the barriers to doctoral studies in ICT, in order to better understand why this is a relatively unattractive option to many potentially suitable graduates when compared to the situation in other disciplines.

Previous research has identified five factors influencing decisions to embark on doctoral studies across all faculties: family and friends, intrinsic motivation, lecturer influence, research experience, and career progression (Guerin et al., 2014). In Engineering more specifically, the reasons for continuing into a PhD are based on a genuine interest in the topic, often inspired by positive undergraduate experiences of engaging with active research (Guerin and Ranasinghe 2010, Jiang and Loui 2012). Baytiyeh and Naja (2011) identify professional attitude, social attitude, financial attitude and subjective norm as factors influencing choices regarding PhD study for engineering graduates; Jiang and Loui (2012) add the sense of attachment to the university department as another important influence in this decision making. While a clearer picture about the motivations underpinning students' decisions to undertake research degrees is starting to develop, very little has been published relating to the *barriers* to continuing study. Although Crede and Borrego (2011) comment briefly on barriers for Engineering students more generally, reliable information relating to ICT specifically appears to be virtually non-existent. As has been found when attempting to encourage greater participation in undergraduate education, understanding these barriers is important for policy makers and universities to develop appropriate strategies for reducing or removing them (Gorard 2006). A recent review of research into access to doctoral education reveals that there is little research into the barriers to students continuing to postgraduate degrees of any sort (McCulloch and Thomas 2013) and this current project, involving students at three Australian universities, goes some way towards addressing that knowledge gap through a specific focus on ICT.

2 Method
The current paper asks: what are the barriers for ICT students moving into study for a research degree? To the best of our knowledge, there is no relevant questionnaire readily available to conduct our investigation. Therefore, a questionnaire was designed to identify the level of interest in pursuing a research degree and the barriers/motivations relevant to those decisions amongst current undergraduate and Honours/Masters students.

2.1 The Questionnaire
The complete questionnaire contained three main sections. Section 1 contained four statements regarding students' intentions after completing their current degree, that is, whether they intended to leave higher education or continue studying (in a different undergraduate degree, in a Masters by coursework degree, or in a research degree). Section 2 contained 13 statements regarding possible barriers to undertaking a research degree. The third section contained 17 statements regarding possible motivations for undertaking a research degree.

Respondents were asked to answer Section 1 and either Section 2 or 3. In each section, they were asked to respond to statements on a 7-point Likert scale with responses ranging from 1 (strongly disagree) to 7 (strongly agree). In addition to the closed questions, respondents were also invited to provide qualitative

		University One		University Two		University Three		Total	
		Post-course intentions % n=99	Barriers to doctoral study % n=79	Post-course intentions % n=45	Barriers to doctoral study % n=37	Post-course intentions % n=21	Barriers to doctoral study % n=17	Post-course intentions % n=165	Barriers to doctoral study % n=133
Gender	Male	88	89	84	86	90	88	87	88
	Female	12	11	16	14	10	12	13	12
Nationality	Australian	47	37	60	59	89	88	55	56
	International	53	63	40	41	21	12	45	44
Age	21-25	76	80	74	79	67	71	74	78
	26-30	19	16	20	16	14	6	19	15
	Over 31	5	4	6	5	19	23	7	7
Level of Study	Final Year U/grad	37	37	67	76	62	71	48	52
	Honors/Masters	63	63	33	24	38	29	52	48

Table 1: Descriptive statistics for respondents

comments at the end of each section. The focus of this paper is on responses to Section 2, which investigated the reasons students identified for not choosing to continue into research degrees.

2.2 Questionnaire Design

Since there is no existing research that focuses directly on the barriers for pursuing a research degree, we used related studies to develop a questionnaire. The main sources for this are Park et al. (2010), who have discussed barriers to undertaking research degrees in medical science. We also used the insights of Naturalistic Decision Making (NDM) (Klein, 2008), particularly in relation to the influence of "past experience" in decision making. Finally, the researchers' domain knowledge gained from extensive experience of the sector was used to inform the questionnaire design. In summary, the questionnaire items were based on five main themes: 1) Financial reasons (4 questions); 2) Attitude (2 questions); 3) Value for degree (2 questions); 4) Lack of awareness (3 questions); and 5) Past experience (2 questions). Participants were invited to indicate the strength of the influence of each element on a 7-point Likert scale, ranging from 1 (not at all) to 7 (a lot). Respondents were also invited to provide comments at the end of each section. Respondents were invited to answer Section 1 and either Section 2 or 3. Here we report on the reasons students identified for choosing not to continue into research degrees.

2.3 Survey Administration and Participants

The three universities involved in this study have different histories, and different strategic and research priorities. They represent three different types of universities: University One is a member of the Group of 8 (Go8) leading research-intensive universities; University Two is part of the Australian Technology Network (ATN) that focuses on the practical application of tertiary education; and University Three is an Innovative Research University (IRU), a collaboration that comprises research universities established more recently than the Go8 group.

Human Research Ethics approval was granted by each of the three universities and hard copies of the survey were handed out in ICT final year undergraduate and Honours/Masters classes. Altogether 172 responses were received. All the respondents answered Section 1 regarding intentions following graduation, and 136 respondents answered Section 2 regarding barriers to undertaking a research degree.

2.4 Overview of the Analysis

Two approaches have been taken to analysing the data gathered in the survey. Firstly, we conducted an overall analysis of the responses (regarding post-course intentions and barriers for pursuing a research degree) using descriptive statistics and then explored the reasons for those decisions in closer detail according to differences between university types (Go8, ATN and IRU). Secondly, an Exploratory Factor Analysis was undertaken to investigate the underlying structure of factors that are perceived by students to be potential barriers to continue studying in a research degree.

2.5 Preliminary Analysis: Descriptive Statistics

An initial evaluation of the dataset resulted in the elimination of seven respondents who had completed less than 75% of the questionnaire. For all other respondents, missing scale items were imputed by determining the mean for the items on the scale (an appropriate data replacement strategy when less than 5% of data is missing) (Tabachnick & Fidell, 2007). No outliers were found for Section 1; one outlier was detected and removed from Section 2. Descriptive statistics regarding demographic characteristics of all respondents, including details of gender, nationality, age, current university and levels of study, are presented in Table 1. We have

		Mean	Std. Deviation	Std. Error Mean	Mode		Overall (n=165)	University One Go8 (n=99)	University Two ATN (n=45)	University Three IRU (n=21)
S1	I want to leave the University and get a job	5.54	1.87	0.15	7	%Disagree	15	15	13	14
						%Neutral	7	5	11	10
						%Agree	78	80	76	76
S2	Continue studying in a different undergrad course	2.39	1.63	0.13	1	%Disagree	84	72	87	76
						%Neutral	5	14	9	10
						%Agree	12	14	4	14
S3	Continue studying in a Masters by course work degree	3.23	1.71	0.13	1	%Disagree	62	51	60	67
						%Neutral	13	25	16	5
						%Agree	25	24	24	29
S4	Continue studying in research degree	3.58	1.82	0.14	4	%Disagree	57	48	60	57
						%Neutral	16	24	16	10
						%Agree	27	27	24	33

Table 2: Post-course intentions

interpreted responses of 5, 6 and 7 as indicating broad agreement with the statement, whereas 1, 2 and 3 indicate broad disagreement.

The data satisfy the assumption of homoscedasticity, therefore *t*-tests could be carried out. In line with the central limit theorem, means of samples from a population with finite variance approach a normal distribution regardless of the distribution of the population. Provided the sample size is at least 30, we can assume that sample means are normally distributed. Given our smallest sample size for a *t*-test is 44, assumptions of normality are satisfied.

2.6 Exploratory Factor Analysis

Exploratory Factor Analysis (EFA) was used to investigate the underlying structure of factors that are perceived by current final year undergraduate and Honours/Masters students to be potential barriers to continue studying in a research degree. EFA is used to reduce a large number of variables into a smaller set of variables (also referred to as factors) and, as its name suggests, it is exploratory in nature and has the advantage of having no expectations of the number or the nature of the factors. Therefore, it is not expected that the themes identified in the questionnaire development stage would necessarily emerge as distinct factors in the EFA. Nevertheless, the results obtained through EFA enable identification of the most important factors for not continuing into a research degree.

3 Post-course intentions

Table 2 shows the respondents' intentions following completion of their current undergraduate/Masters degree with most (78%) intending to leave the university system for employment after completion of their current degree. The most common response for this statement was 7 on the Likert scale. The statement "continue studying in a different undergraduate degree" received the lowest percentage for broad agreement (12%) with the most common response being 1 on the Likert scale. "Continue studying in a Masters coursework degree" (25%) and "continue studying in a research degree" (27%) received similar levels of broad agreement. However, these options were not interpreted by the respondents as being absolutely mutually exclusive, demonstrating the potentially fluid nature of post-graduation decision-making, with 13% of respondents being in broad agreement with both the possibility of pursuing a Masters by coursework degree and also the possibility of pursuing a research degree. This fluidity is also demonstrated by the fact that 17% of respondents were in broad agreement with both "I want to leave the university and get a job" and "Continue studying in a research degree". Fluidity in intention (and thus decision-making) is something that comes through the results fairly consistently and has implications for both policy makers and university administrators.

Table 2 also shows that the majority of the students in all three institutions plan to leave the university and find a job after completing their current degree (S1). Interestingly, while 14% of students from both the Go8 and IRU universities broadly agreed to the possibility of continuing studying in a different undergraduate course (S2), just 4% of students from the ATN university indicated this intention. Although the responses here are from institutions with different emphases on research,

		Mean	Std. Deviation	Std. Error Mean	Mode		Overall (n=133)	University One (n=79)	University Two (n=37)	University Three (n=17)
						%Disagree	12	15	8	6
S1	I want to start earning money	5.71	0.15	1.69	7	%Neutral	3	4	3	0
						%Agree	85	81	89	94
						%Disagree	47	46	49	53
S2	I don't expect high enough grades	3.55	0.17	2.00	1	%Neutral	19	21	22	0
						%Agree	34	33	30	47
						%Disagree	66	59	54	65
S3	I never thought about it	3.26	0.15	1.73	2	%Neutral	13	24	16	12
						%Agree	21	17	30	23
						%Disagree	41	46	35	29
S4	I don't really know what it would involve	3.90	0.14	1.66	5	%Neutral	21	21	27	6
						%Agree	38	33	38	65
						%Disagree	56	65	38	59
S5	I don't know anything about research	3.38	0.15	1.69	2	%Neutral	13	10	22	6
						%Agree	31	25	41	35
						%Disagree	41	40	41	47
S6	I think research will be boring	3.81	0.15	1.76	5	%Neutral	20	19	27	12
						%Agree	38	41	32	41
						%Disagree	41	46	32	41
S7	I think research would be too difficult	3.74	0.14	1.59	4	%Neutral	26	20	35	35
						%Agree	32	34	32	24
						%Disagree	20	24	11	18
S8	I'm tired of studying and want a change	5.11	0.15	1.75	6	%Neutral	13	15	8	12
						%Agree	68	61	81	70
						%Disagree	32	33	24	41
S9	I think fees might be too expensive	4.20	0.16	1.86	4	%Neutral	31	33	30	24
						%Agree	38	34	46	35
						%Disagree	38	37	35	53
S10	I don't want to add to my fee/help debt	4.15	0.17	1.95	4	%Neutral	18	20	19	6
						%Agree	44	43	46	41
						%Disagree	24	29	19	12
S11	I think scholarships are too small to live on	4.14	0.13	1.49	4	%Neutral	44	39	49	53
						%Agree	32	32	32	35
						%Disagree	38	38	32	47
S12	I think employers don't want people who are too highly qualified	3.78	0.15	1.76	4	%Neutral	32	33	27	41
						%Agree	30	29	41	12
						%Disagree	38	41	27	47
S13	I don't want to work in a university, so do not need a PhD	3.82	0.16	1.79	4	%Neutral	28	24	32	35
						%Agree	35	35	41	18

Table 3: Barriers to pursuing a research degree

similar percentages of students intended to continue studying in a research degree (S4): University One (Go8) 27%, University Two (ATN) 24% and University Three (IRU) 33%. Not all have decided on their next step, though: 24% of students from University One, 16% from University Two and 10% from University Three remain uncertain about pursuing a research degree. While such uncertainty in ICT areas is considerably less than that found by Shaw et al. (2013) in a cross-institutional study of Honours students in all faculties (which stood at one third of students), this level of uncertainty supports the earlier contention that the decision-making process regarding research degrees is not static but fluid.

4 Barriers to doctoral study

The reasons for deciding not to continue studying in a research degree are shown in Table 3. The most common reason given was the desire to start earning money (S1) with 85% of respondents being in broad agreement with this statement, and most commonly representing this at 7 on the Likert scale. This was followed by being tired of studying and wanting a change (67% of respondents), which most commonly scored 6.

The least common response in this section was "I never thought about it" (21% in broad agreement), while two thirds (66%) were in broad disagreement with this statement. This suggests that many of the respondents have considered embarking on a research degree, but decided against it. Interestingly, 38% of respondents expressed broad agreement with the statement "I don't really know what it would involve" and that 31% of respondents were in broad agreement with "I don't know anything about research". Perhaps students do not consider the possibility of undertaking research because of a lack of knowledge or understanding about what form that might take in ICT.

4.1 Effect of Institution

Comparing results between universities reveals some interesting patterns (Table 3). The most common barrier to pursuing a research degree for all institutions was overwhelmingly the desire (or need) to earn money, in line with the findings of Crede and Borrego (2011) that financial reasons pose a major barrier to postgraduate study in the US context. Some students (especially those from ATN University Two) also seem to believe that employers may not want people who are too highly qualified (S12). A higher percentage (65%) of the IRU University Three students were in broad agreement that they do not know what research would really involve compared to the students of Go8 University One (33%) and students of University Two (38%); in a possibly related finding, a higher proportion of University Three students thought their grades would not be high enough to allow them to pursue doctoral study. When added to those students at Universities Two and Three who report a lack of knowledge about research being a barrier, this may indicate that options for postgraduate study in ICT are not presented effectively at any of the three institutions studied here, regardless of the broader institution's priorities regarding research. Furthermore, 80% of students at the ATN, 70% of the students at the IRU and 61% of students at the Go8 university were in broad agreement that they are tired of studying and want a change.

5 Factors influencing decisions not to pursue research degrees

In seeking to uncover the underlying structure of the barriers to undertaking a research degree, we conducted an Exploratory Factor Analysis. This involves a series of sequential steps (e.g., selection of the number of factors, selection of the factor rotation method) that also involve evaluating multiple options. This procedure is explained in detail in our previous work (Guerin et al., 2014).

Although sample size is important in factor analysis, there is no agreement as to the optimum or minimum number and a variety of opinions can be found in the literature. Hair et al. (1995) suggest that sample sizes should be 100 or greater. For Comrey and Lee (1992), 200 is seen as a fair sample size. However, MacCallum et al. (1999) take the view that such rules of thumb can be misleading, explaining that they often fail to take into account the complex dynamics of a factor analysis. As an example, when communalities are high (greater than .60) and each factor is defined by several items, appropriate minimum sample sizes can actually be relatively small (Henson, 2006). In our study, as presented in Table 4, most of the communality values are greater than 0.6. It is also worth noting that Sapnas and Zeller (2002) point out that even as few as 50 cases may be adequate for factor analysis.

The ratio of subject-to-variable is an important aspect to be considered before conducting an EFA. When total sample size increases, this ratio becomes less important; on the other hand, the subject-to-variable matters more when the sample size is relatively low (Osborne & Costello, 2004). Further, for a large sample size or large ratio, the results will be more reliable (Osborne & Costello, 2004). In our study, even though the sample size was 133, a significant case-to-variable ratio of approximately 10:1 was present, allowing us to make strong claims from the data.

The correlation matrix was inspected for correlations in excess of 0.3. The literature warns that, if no correlation exceeds this, the applicability of factor analysis should be reconsidered (Tabachnick & Fidell, 2007). The Kaiser-Meyer-Olkin measure of sampling adequacy tests whether the partial correlations among variables are small and this was 0.65, above the recommended value of 0.6 (Hair et al., 2009). Bartlett's test of Sphericity tests whether the correlation matrix is an identity matrix, hence can be used to determine whether the factor model is appropriate. This value was significant ($p<0.05$) (c^2 = 443.1, df=78, Sig.=0.000), indicating the possibility of using factor analysis with the data.

For the 13 items used in the questionnaire, a Principal Component Analysis (PCA) was conducted. To determine the number of factors to retain, we used Parallel Analysis (PA). In recent research, PA is often recommended as the best method to assess the number of factors (Lance, 2006; O'Connor, 2000; Velicer et al., 2000). PA takes into account sampling error and retains factors when actual eigenvalues surpass random ordered eigenvalues. Parallel Analysis indicated that four factors should be retained. Initially, the four factors accounted for approximately 61% of the total variance; this is in line with the heuristic recommended by Hair et al. (2009), which states more than 50% of the variance should be explained by the retained factors.

Factor rotation maximises high item loadings and minimises low item loadings, therefore producing a more interpretable and simplified solution. As suggested by Tabachnick and Fiddell (2007), we undertook an oblique rotation first and inspected the correlation of factors. Since no correlation exceeds the threshold of 0.32,

Items	1	2	3	4	h^2
I think fees might be too expensive	0.877				0.789
I don't want to add to my fee/help debt	0.871				0.759
I think scholarships are too small	0.756				0.631
I don't really know what it would involve		0.832			0.732
I don't know anything about research		0.854			0.763
I think research would be too difficult		0.602			0.399
I want to start earning money			0.734		0.714
I'm tired of studying and want a change			0.884		0.809
I think employers don't want people who are too highly qualified				0.734	0.629
I don't want to work in an university, so do not need a PhD				0.811	0.699

Table 4: Factor loadings (EFA through the principal component analysis with varimax rotation).

Notes: values < 0.4 are suppressed; Abbreviations: h^2 = Communality; Factor 1=Finance; Factor 2=Perceptions of research; Factor 3=Desire for change; Factor 4=Career orientation.

varimax rotation was used. To further simplify interpretation and develop an efficient measure, only those items that loaded highly and uniquely on each factor were retained. Thus, we omitted items that loaded less than 0.4 on all the factors and the items that cross-loaded on more than one factor. Item 2 (I don't expect high enough grades to go on to a research degree) failed to load highly on any of the factors. Item 3 (I never thought about it) and item 6 (I think research would be boring) cross-loaded on more than one factor.

Four factors emerged from the analysis: 1) Finance; 2) Perceptions of research; 3) Desire for change; and 4) Career orientation. The rotated component matrix and the communality values are presented in Table 4. The four factors accounted for 26%, 18%, 13%. and 11% of the total variance, respectively. Internal consistency was measured using Cronbach's alpha (Cronbach, 1951). The alpha values for the factors were 0.8, 0.7, 0.6, 0.5, respectively. Cronbach's alpha is grounded on the theory of the 'tau equivalent model' which assumes that each test item measures the same latent trait on the same scale (Tavakol & Dennick 2011). The alpha value can be over- or under-estimated based on the test length. Our previous work (Guerin et al. 2014) explains the dynamics of Cronbach's alpha value in detail. Loewenthal (2001) has stated that a high alpha level is unlikely with a small number of items (test length of Factor 3 and Factor 4 is two). Nevertheless, we can consider accepting lower alpha values if there are good theoretical and/or practical reasons for all items in a given dimension, and the number of items in that dimension is small (less than about 10 items).

Following an exploratory factor analysis, factor scores may be computed and used in subsequent analyses. A factor score is a numerical value that is meant to indicate a person's relative spacing or standing on a latent factor. Therefore, factor scores were computed for every participant based on Bartlett factor coefficients. The Bartlett method is considered to be a redefined method of computing factor scores. Redefined methods aim to maximize validity by producing factor scores that are highly correlated with a given factor and also attempt to maintain the existing relationships between factors. In order to identify whether there are any significant differences among the four factors and to identify the most important factor, repeated measures ANOVA (Analysis of Variance) was carried out.

Basically, ANOVA provides a statistical test to determine if the means of several groups are equal or not. This can be seen as a generalization of the t-test for more than two groups. The reason for carrying out a repeated measures ANOVA test as opposed to multiple t-tests is as follows. Every time one conducts a t-test there is chance of making a type 1 error that corresponds to the confidence interval. Therefore, when more hypothesis tests are carried out, there can be more risk of making a Type 1 error and the power of the test can be significantly reduced. However, the ANOVA test controls these errors and the Type 1 error remains at 5%.

Repeated measures of ANOVA indicated significant differences among the four factor scores ($F(3,396)=39.41$, $p<0.05$). The 'Change orientation' was shown as the most important for the participants (mean=5.7). This was followed by the 'Perception of research' factor (mean=4.3), 'Financial Factor' (mean=4.1) and the 'Career Orientation Factor' (mean=3.6). However, repeated measures of ANOVA do not indicate where these differences occur exactly. Therefore, we conducted a post hoc test using the Bonferroni technique which indicated significance ($p<0.05$) differences between 'Change Orientation' and all other factors.

6 Discussion and Conclusions

The results reveal interesting variations and similarities across the university groups considered in this study. The large majority of students surveyed across the three universities intended to leave the university system and find a job on completing their current undergraduate or Masters degree. Their readiness in expecting to be able to get a job suggests that these students believe their qualifications will be adequate and that there are reasonable employment prospects available to them. Indeed, some evidently think that a further qualification might render them less attractive in the eyes of some employers. However, around a quarter did express an interest in continuing into a research degree, the highest proportion of these being at the Innovative Research University institution (University Three), with the Australian Technology Network and Group of Eight institutions having lower (but similar) proportions considering this option.

There was also a group of students (17%) who were interested in both leaving university and starting to earn and also possibly pursuing a research degree. Two important tentative conclusions can be drawn from this finding. Firstly, these may be the individuals who are most likely to come back to study after a period of working in industry (Baytiyeh and Naja (2011). If this is so, the data suggest that universities would be wise to create easy pathways for such "returners" (Peters and Daly 2013) to re-enter the university system as doctoral candidates; universities should also actively promote this possibility to undergraduates and coursework Masters students.

The second conclusion (and this is supported by a broader range of data as is reported earlier in the paper) is that students' decision-making about undertaking a research degree is something which is fluid rather than fixed. There is evidence in the responses that some students are not interested in undertaking a research degree because they are not sure what 'research' involves. Approximately one third of respondents expressed uncertainty about the form this could take in ICT, and we believe students would benefit from hearing more about their lecturers' own research experience and research projects, as well as the cutting-edge research being undertaken in their areas. Again, the EFA identified that ICT students' perception of research was an important barrier to choosing the research pathway on completing their current degrees. Other studies have shown that positive undergraduate experiences of research can influence the choices students make in this regard in related fields (Guerin and Ranasinghe 2010). Because students are expressing a degree of fluidity in their decision-making regarding research degrees, universities would be well advised to make sure that undergraduates understand the nature of research in their disciplines, think of research as a legitimate career path, and know how to pursue such a course of action. The widening participation discourse has promoted this approach in undergraduate education; it is time to apply these insights to doctoral study (McCulloch and Thomas 2013). This type of activity would also contribute to strengthening the teaching–research nexus that has been the subject of considerable discussion in higher education over the last decade (Jenkins *et al.* 2003, Barnett 2005, Brew 2006, Healey and Jenkins 2006, Simons and Elen 2007, Verburgh *et al.* 2007, Trowler and Wareham 2008, Brew 2010).

To conclude, the vast majority of students in ICT want to move into the workforce on completing their degrees rather than continuing into research degrees. This may be motivated largely by a desire to start earning money, but there is evidence here (mirroring that found by Crede and Borrego (2011)) that many also find their courses demanding and feel that they need a break from study. Many identify that they are tired of studying and want a change. This is reflected by the high means as a whole in Table 3 (I want to earn money at 5.71; I'm tired of studying at 5.11) and is further supported by the EFA that not only links these two elements as one of the factors, but also indicates that this is the most important factor in the decision-making of this group.

Nevertheless, there is clearly a substantial group who are interested in pursuing a research degree after a break from higher education; ICT departments should make it clear to undergraduate students that this is a possibility, and should also find ways to create smooth pathways back into study for this group. This is particularly important in view of the fluidity in decision-making that we have identified here.

The experiences students have of research during their undergraduate study may be the inspiration that brings them back to study later in their careers. If research can establish what motivates ICT students to continue their studies, including what kinds of undergraduate experience of the teaching–research nexus might influence their decisions, we will be in a good position to support greater numbers of students to pursue research degrees in ICT.

7 Acknowledgements

The authors wish to gratefully acknowledge the support of the Australian Council of Deans of ICT (ACDICT) Learning and Teaching grant (ALTA) which made this research possible.

8 References

Barnett, R. (Ed.). (2005). *Reshaping the university: New relationships between research, scholarship and teaching*. McGraw-Hill Education/Open University Press.

Baytiyeh, H. and Naja, M.K. (2011). Contributing factors in pursuit of a phd in engineering: The case of Lebanon. *The International Journal Of Engineering Education*, 27 (2): 422–430.

Brew, A. (2006). *Research and teaching*. Palgrave Macmillan.

Brew, A. (2010). Imperatives and challenges in integrating teaching and research. *Higher Education Research & Development*, 29 (2): 139–150.

Comrey, A.L. and Lee, H.B. (1992). *A first course in factor analysis*. L. Erlbaum Associates.

Crede, E. and Borrego, M.J. (2011). Undergraduate engineering student perceptions of graduate school and the decision to enroll. *American Society for Engineering Education*.

Cronbach, L.J. (1951). Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3): 297–334.

Gorard, S. (2006). *Review of widening participation research: Addressing the barriers to participation in higher education: A report to HEFCE by the University of York, higher education academy and institute for access studies*: HEFCE.

Graduate Careers Australia (2013). GCA home page [online]. Graduate Careers Australia. Available from: http://www.graduatecareers.com.au [Accessed Access Date 24 August 2014].

Guerin, C. and Ranasinghe, D. (2010). Why I wanted more: Inspirational experiences of the teaching-research nexus for engineering undergraduates. *University Teaching & Learning Practice*, 7(2) Available: http://ro.uow.edu.au/jutlp/vol7/iss2/8.

Guerin, C., Jayatilaka, A. and Ranasinghe, D. (2014). Why start a higher degree by research? An exploratory factor analysis of motivations to undertake doctoral studies. *Higher Education Research & Development (HERD)*. DOI: 10.1080/07294360.2014.934663..

Hair, J.F., Black, W.C., Babin, B.J., and Anderson, R.E. (1995; 2009). *Multivariate data analysis* (7th ed.). Prentice Hall.

Healey, M. and Jenkins, A. (2006). Strengthening the teaching-research linkage in undergraduate courses and programs. *New Directions for Teaching and Learning*, 107, 43–53.

Henson, R.K. (2006). Use of exploratory factor analysis in published research: Common errors and some comment on improved practice. *Educational and Psychological Measurement*, 66(3), 393–416.

Jenkins, A., Breen, R. and Lindsay, R. (2003). *Reshaping teaching in higher education: Linking teaching with research*. London: SEDA and Routledge.

Jiang, A. and Loui, M.C. (2012). What should I do next? How advanced engineering students decide their post-baccalaureate plans. Frontiers in Education Conference (FIE). Seattle, 3-6 October 2012: IEEE, 1-6.

Klein, G. (2008). Naturalistic decision making. Human Factors. *The Journal of the Human Factors and Ergonomics Society*, 50(3): 456–460.

Lance, C.E. (2006). The sources of four commonly reported cutoff criteria: What did they really say? *Organizational Research Methods*, 9(2), 202–220.

Loewenthal, K.M. 2001. *An introduction to psychological tests and scales*. Psychology Press.

O'Connor, B.P. (2000). SPSS and SAS programs for determining the number of components using parallel analysis and velicer's MAP test. Behavior Research Methods, Instruments, & Computers. *Journal of the Psychonomic Society*, 32(3), 396–402.

Osborne, J.W., and Costello, A.B. (2004). Sample size and subject to item ratio in principal components analysis. *Practical Assessment, Research & Evaluation*, *9*(11).

MacCallum, R. C., Widaman, K. F., Zhang, S., and Hong, S. (1999). Sample size in factor analysis. *Psychological Methods*, 4(1): 84–99.

McCulloch, A. and Thomas, L. (2013). Widening participation to doctoral education and research degrees: A research agenda for an emerging policy issue. *Higher Education Research & Development*, 32(2): 214–227.

Osborne, J.W., and Costello, A.B. (2004). Sample size and subject to item ratio in principal components analysis. *Practical Assessment, Research & Evaluation*, 9(11).

Peters, D.L. and Daly, S.R. (2013). Returning to graduate school: Expectations of success, values of the degree, and managing the costs. *Journal of Engineering Education*, 102(2), 244–268.

Sapnas, K. G., and Zeller, R. A. (2002). Minimizing Sample Size When Using Exploratory Factor Analysis for Measurement. *Journal of Nursing Measurement*, 10(2): 135–154.

Shaw, K., Holbrook, A. and Bourke, S. (2013). Student experience of final-year undergraduate research projects: An exploration of 'research preparedness'. *Studies in Higher Education*, 38(5): 711–727.

Simons, M. and Elen, J. (2007). The 'research–teaching nexus' and 'education through research': An exploration of ambivalences. *Studies in Higher Education*, 32(5): 617–631.

Tabachnick, B.G., & Fidell, L.S. (2007). *Using multivariate statistics* (5th ed.). Boston, MA: Allyn and Bacon.

Tavakol, M., & Dennick, R. (2011). Making sense of Cronbach's alpha. *International Journal of Medical Education*, 2: 53–55.

Trowler, P. & Wareham, T., 2008. *Tribes, territories, research and teaching enhancing the teaching-research nexus*. York: The Higher Education Academy.

Velicer, W., Eaton, C., & Fava, J. (2000). *Construct explication through factor or component analysis: A review and evaluation of alternative procedures for determining the number of factors or components.* New York, NY, US: Kluwer Academic/Plenum Publishers. Retrieved January 19, 2014, from http://ezproxy.library.yorku.ca/login?url=http://search.proquest.com/docview/619671749?accountid=15182

Verburgh, A., Elen, J. & Lindblom-Ylänne, S. (2007). Investigating the myth of the relationship between teaching and research in higher education: A review of empirical research. *Studies in Philosophy and Education*, 26(5): 449–465.

Teaching in First-Year ICT Education in Australia: Research and Practice

Michael Morgan
Monash University
Australia
michael.morgan@monash.edu

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Matthew Butler
Monash University
Australia
matthew.butler@monash.edu

Katrina Falkner
University of Adelaide
Australia
katrina.falkner@adelaide.edu.au

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Amali Weerasinghe
University of Adelaide
Australia
amali.weerasinghe@adelaide.edu.au

Abstract

This paper details current research and teaching practice for first-year Information and Communications Technology (ICT) students at Australian universities. The project aims to record and disseminate good practice in first-year ICT teaching in Australia. The aim of the paper is to examine how academics are addressing the challenge of engaging first-year ICT students in the learning process. Two sources of data are used, a systematic survey of research literature from the last five years and detailed interviews of 30 academics involved in first-year teaching duties. Academics from 25 Australian universities represented a range of universities, including six from the Go8 group, three from the ATN group, and five from the IRU group. The paper highlights current areas of research, any gaps in the research literature, examples of current good teaching practices, and recommendations for further research.

Keywords: First Year; Student Experience; Teaching.

1 Introduction

This paper presents a survey of current research literature and current practice in Australian universities for the teaching of first-year ICT students. It is motivated by the unique challenges facing ICT educators as they design and deliver educational experiences for first-year students in the ICT domain. The challenges faced by ICT students in the transition from secondary education are evidenced by the relatively high rate of attrition in ICT courses, a reduced engagement in on-campus learning experiences and a perceived lack of relevance to some potential student groups (Sheard, Carbone, & Hurst, 2010). In a search of the literature we found few examples that addressed these issues in the ICT context and in the Australian setting. While a lot of worthwhile research is being conducted into specific teaching practices in specific contexts, there is a need to properly collate and review this research in order to drive change in practice more broadly across the Australian Higher Education sector.

To investigate current research and practices in first-year ICT courses in the Australian context, the authors investigated six broad themes that together describe the learning experience: "what we teach", "where we teach", "how we teach", "how we assess", "learning support" and "student support". Only the "how we teach" theme is presented in this paper due to space considerations. Within this theme the different aspects of teaching are discussed in relation to issues such as student engagement, student retention, learning outcomes and broadening the relevance of ICT courses to a wider range of students.

2 Research Approach

The research team (the authors of this paper) designed two phases for this project: a review of research literature from the last five years, and interviews of academics involved in the delivery of first-year programs to survey current practice. A detailed description of the methodology used in this project is reported in *Experiences of first-year students in ICT courses: good teaching practices*: Final Report: ICT student first year experiences (http://www.acdict.edu.au/ALTA.htm). Accordingly, a brief summary is presented below, with focus placed on the "how we teach" theme.

In phase 1 a systematic review was conducted of the literature from 2009 to 2014 in the area of computing education. Keyword searches were carried out in Google Scholar and the IEEE Xplore and ACM Digital Library databases, along with manual searches of key computing education journals and conference proceedings.

In phase 2, semi-structured phone interviews were conducted with academics from Australian universities between February and March 2014. Participants were identified as key staff involved with the design and/or delivery of ICT courses to first-year students. Thirty academics from 25 Australian universities were interviewed. These included six Group of Eight (Go8), three Australian Technology Network (ATN), six Innovative Research (IRU) universities and three Regional University Network (RUN). The interviews averaged 53 minutes, with detailed notes being taken. They were audio recorded so that relevant comments could be transcribed at a later time. The interview script focused on six key themes and all interviewees were sent the interview questions before the interview. Questions were devised to elicit responses about initiatives in teaching practice; for example, "Do you use any 'novel' teaching practices, such as peer instruction, flipped classroom, students contributing to the learning of others, e.g. through Peerwise, student seminars, etc?". Follow up questions on specific issues were also asked where appropriate.

3 How we teach

The investigation of teaching in first-year ICT courses in Australian universities was concerned with all aspects of the design and delivery of university-level learning experiences to first-year ICT students, and associated supporting academic activities. We begin our investigation of teaching in first-year ICT courses with a review of the literature. This gives a broad perspective of assessment in first-year ICT courses during the past five years, highlighting Australian studies. Following this, an analysis of our interviews of academics provides insights into teaching practices in Australian courses.

3.1 Literature Perspectives in ICT Teaching Practice

The systematic literature review identified 57 papers that were considered relevant to the theme of "how we teach", grouped into four main topics:

1. theories and models of teaching and learning
2. approaches to teaching
3. cooperative and collaborative learning
4. social media and learning communities

All papers were set in the higher education sector and in the ICT discipline. Most papers were focused on teaching in first-year courses. Fifty papers (88%) dealt with teaching programming, particularly introductory programming. Eleven were Australian studies.

Theories and models of learning

A number of researchers have explored theoretical bases for teaching and learning in the ICT discipline, all in the context of introductory programming.

An Australian study by Mason and Cooper (2012) investigated lecturers' perceptions of the mental effort required for different aspects of their programming units. Interpreting the findings using cognitive load theory (Sweller, 1999), the authors propose that many low-performance students fail to learn due to cognitive overload. Skudder and Luxton-Reilly (2014) reviewed the use of worked examples in computer science. They evaluated different types of worked examples in terms of the cognitive load on the learner, and recommend example-problem pairs and faded worked examples as most suitable for novices.

A number of researchers have challenged the 'programming gene' view that people are either inherently programmers or have great difficulty picking up programming fundamentals. Robins (2010) investigated possible reasons for the bimodal grade distribution that some believe is typically found in introductory programming courses. He proposes that this is caused by the 'learning edge momentum' (LEM) effect whereby success in learning a concept helps in learning subsequent closely related concepts. In the programming domain, where concepts are tightly integrated, the LEM effect drives students to extreme learning outcomes.

A group of Australian researchers have explored the learning of programming from a neo-Piagetian perspective (Lister, 2011; Corney et al., 2012; Teague & Lister, 2014). From a series of empirical studies they propose that novice programming students pass through neo-Piagetian stages of sensorimotor, preoperational, and concrete operational before reaching the formal operational stage where they can operate as competent programmers. They recommend that introductory programming teachers use a neo-Piagetian perspective in their instruction where they consider the reasoning levels of their students.

A couple of studies have used Dweck's (2000) 'mindset' theory in introductory programming teaching programs. Dweck identified that learners may have 'fixed' or 'growth' mindsets, which have implications for their learning. Students with a growth mindset focus on learning goals and continue to focus on learning, even after failures. By contrast, students with a fixed mindset focus on performance goals, and want to be seen as achieving well at all times. Through several interventions implemented in an introductory programming course, Cutts et al. (2010) found that they were able to shift students from fixed to growth mindsets, resulting in a significant improvement in their learning. An intervention program by Hanks et al. (2009) reported less success.

Dann et al. (2012) report an application of the theory of 'mediated transfer' (Salomon & Perkins, 1988) in the design of an introductory programming course. The purpose was to aid students in transferring their knowledge of programming concepts learnt in Alice 3 to the Java context. Using this approach they found dramatic improvement in students' final exam performances.

A couple of papers report the use of Biggs' model of 'constructive alignment' (Biggs, 1996) as a framework for design of introductory programming units. Thota and Whitfield (2010) and Australian researchers Cain and Woodward (2012) describe the design of their courses and present results from action research studies. They discuss the implications of the use of constructive alignment as a framework for course design.

A comprehensive review by Sorva (2013) summarises the research on challenges faced by novice programmers in understanding program execution. Based on findings, he proposes that the 'notional machine' should be used explicitly in introductory programming to help novices understand the runtime dynamics of programs. Ma et al. (2011) investigated novice students' mental models of programming concepts, finding that many held non-viable mental models of key concepts. Through a teaching approach using visualisation of program execution they found that they could challenge and change students' misconceptions and help them develop a better understanding of key concepts.

Approaches to teaching

Different approaches to teaching form a broad topic encompassing the use of techniques, tools, technologies and games in teaching first-year ICT courses.

1. Teaching Techniques

A variety of teaching techniques for first-year ICT courses were found, all but one in the context of programming. These were typically introduced to improve students' skills and knowledge of a particular learning outcome and/or to motivate and engage students in the learning process.

Caspersen and Kölling (2009) present STREAM, a programming process for novice programmers. This process was derived from a stepwise improvement framework that the authors developed by unifying current good practices in software development. STREAM has been used in two universities, and a study indicates that it helped in the development of students' software development competencies.

Apiola, Lattu & Pasanen (2012) present CSLE (Creative-Supporting Learning Environment), a theoretical framework for designing a course to support students' creative activities. The framework was trialled with a programming course using robotics, and an evaluation indicated that students gained many creative experiences during the course.

Hu, Winikoff & Cranefield (2012; 2013) describe an approach to teaching introductory programming using the concepts of 'goals' and 'plans'. They propose a notation and a programming process incorporating these concepts. An evaluation of the approach using an experimental method indicates a positive improvement in students' performance in their programming exam.

Pears (2010) discusses the concept of program quality and students' conceptions of program quality. He describes an approach used in an introductory computing course designed to give students an understanding of program quality. An assessment of student code produced for their project work indicated a level of quality above what is normally produced by first-year students.

Hertz and Jump (2013) present 'program memory traces', a paper-based approach for code tracing that models program execution in the computer's memory. A study of the use of this approach in an introductory programming class showed improvement in students' programming ability, decrease in dropout rates and significant improvement in students' grades.

The only example found outside the programming context was NEMESIS (Marsa-Maestre et al., 2013), a framework for generating scenarios for teaching network and security systems. An evaluation of the framework with a first-year Internet security systems course found that the students and teachers were positive about the use of the framework and the scenarios generated.

2. Games

Game-based learning and assessment tasks are often used to motivate and engage students in the learning process. Eagle and Barnes (2009) and Morazán (2010) describe their use of games in introductory programming courses. They report findings of studies that show that learning activities based on games are useful tools to interest and enthuse students in programming. However a study of the use of mobile games by Kurkovsky (2013) found mixed results in terms of student engagement and motivation.

Bayzick et al. (2013) present ALE (AndEngine Lehigh Extension), a platform for Android game development. ALE emphasises code reading before students attempt code writing. Experiences with using the platform in an introductory programming course found that students responded positively to the tool and wrote "compelling mobile games in under 18 hours" (p.213).

3. Tools and technologies

A range of tools and technologies have been developed or adapted for use in computing education, all but one in the context of programming.

Anderson and Gavan (2012) report on the introduction of LEGO Mindstorms NXT into an introductory programming course. They found that students' results on assignment work and exams improved, and concluded from a student evaluation that the activities were a stimulating and engaging challenge for the students. Apiola, Lattu & Pasanen (2010) also describe a programming course that uses LEGO Mindstorms robotic activities. On the basis of many positive student comments during and after the course, the authors argue that robots are powerful tools for motivating students.

These conclusions were not supported by a study by McWorter and O'Connor (2009) who used the Motivated Strategies for Learning questionnaire to assess the effect of LEGO Mindstorms robotic activities on student motivation in an introductory programming course. An experimental study showed no difference in intrinsic motivation between the students using LEGO and non-LEGO activities, although responses to qualitative questions indicated that some of the LEGO students enjoyed the activities.

Summet et al. (2009) describe an introductory programming course where each student is provided with a pre-assembled robot which is used as the teaching context. Results of a comparative study showed that the robot class students gained significantly higher results than the non-robot class students.

Daniels (2009) reports on an application of Nintendo Wii Remote (wiimote) technology in an introductory computer engineering and problem-solving class, and the laboratory exercises designed to use the technology. Following a study of the use of the technology, the authors believe that the activities helped students achieve the core learning objectives of the course and that student engagement improved.

A common application of technology in computing education is program or algorithm visualisation, which is used to clarify and explain concepts.

Sorva, Karavirta & Malmi (2013) reviewed visualisation systems designed to help introductory computing students understand the runtime behaviour of computer programs. Evaluations of the systems provided indicate that they are generally useful in helping students learning programming; however, the influence on learner engagement is not clear.

Pears and Rogalli (2011) present an extension to the widely used program visualisation tool Jeliot, where students are able to receive and respond to Jeliot-generated questions on their mobile phones. They propose that this can be used interactively in a lecture, providing an alternative to clicker technology.

Australian researchers Heinsen Egan and McDonald (2014) describe systems for visualising runtime memory state and their integration into the SeeC system. This system will be used initially in a first-year Operation Systems course and the C Programming Language course.

The only example of a tool or technology found outside the programming context was an intelligent tutoring system for learning Rapid Application Development in a database environment. An Australian study by Risco and Reye (2012) describes the Personal Access Tutor (PAT) and an evaluation of the tool in a first-year database course, showing that students and staff found it easy to use and that it was beneficial for students' learning.

Cooperative and collaborative learning

Various teaching approaches have been developed to encourage collaborative and cooperative work behaviour in first-year computing students, often with the aim of developing and fostering learning communities.

Hamer et al. (2012) provide a concise overview of current research perspectives on learning communities by exploring the concept of 'contributing student pedagogy' (CSP). The concept of CSP was developed by Collis and Moonen (2005) who emphasise the process of learning by engaging students as co-creators of learning resources. CSP incorporates social constructivism in a practical manner, combining both content learning and inter-personal skills acquisition in a meaningful way (Hamer et al., 2012, p 315). The learning benefits of engaging learners as active co-creators of the learning experience have been demonstrated in a number of subject domains. Collaborative learning has been used as one of the primary methods of implementing CSP as it requires learners to externalise their understanding in order to work with their peers.

Collaborative learning describes a range of practices where students work in groups sharing knowledge or work on a project. An example of a teaching approach that uses collaborative learning is the 'peer-led team learning' (PLTL) approach as described by Murphy et al. (2011). PLTL involves a small group of students working collaboratively to solve problems. Each group is led by an undergraduate workshop leader who has been specially trained in PLTL techniques. Murphy et al. claim that their PLTL program was highly beneficial for peer leaders, who also benefit from the program as they gain confidence in themselves as computer scientists.

A couple of studies discuss collaborative learning techniques used to increase engagement in lectures. Simon et al. (2010) report on an application of peer instruction (PI) using clicker technology in two introductory programming units. PI is a teaching technique that involves students answering a question on a vote-discuss-revote model. An evaluation found that students were generally very positive about this approach and that the accuracy of the responses increased after a follow-up discussion. The instructor reported value in being able to identify concepts that students had not yet mastered. Kothiyal et al. (2013) describe the implementation of a similar active learning strategy, think-pair-share (TPS), in a large introductory programming class. TPS involves students working on an instructor-led activity individually, then in pairs, and then as a whole class. The authors report levels of student engagement for each activity ranging from 70% to 90%.

Cooperative learning, a specific kind of collaborative learning, is a teaching strategy requiring students to work together to improve their understanding or to complete a task. At an Australian university, Falkner and Palmer (2009) integrated cooperative learning techniques into an introductory computer science course, resulting in increased class attendance, improved learning outcomes and increased student motivation. Beck and Chizhik (2013) report on the implementation of cooperative learning in an introductory computing course and also found an improvement in students' exam results.

Lasserre and Szostak (2011) used a team-based learning (TBL) approach, requiring students to work on exercises in teams. The approach had a positive outcome on student learning: 20% more students completed the course and 20% more students passed the final exam. Informal inspections of the final exam answers suggest that students who learnt using the TBL approach had increased confidence in writing programs. Another team-based approach, reported by Hundhausen, Agrawal & Agrawal (2013), involved peer-reviewing code with the help of a moderator. A series of studies showed that pedagogical code reviews (PCR) facilitated multi-level discussions of code practices, providing opportunities to develop soft skills in introductory computing courses. The study also showed that the online implementation of PCR was not as effective as the face-to-face PCR.

Many studies have investigated the effectiveness of pair programming as a form of cooperative learning for introductory programming students. Pair programming is a programming technique where two people work together to write a program, alternating between 'driver' and 'navigator' roles. Australian researchers Corney, Teague & Thomas (2010) implemented pair programming in an introductory programming course at an Australian university and report that it was well received by students. Wood et al. (2013) describe the use of pair programming in the early weeks of an introductory programming course. Students were paired based on comparable levels of confidence, and it was found that students with the lowest level of confidence performed better working in a pair than individually. Staff observed increased engagement, motivation and performance. Radermacher, Walia & Rummelt (2012) investigated the formation of pairs using Dehnadi's mental model consistency (MMC) test and found evidence supporting the approach of matching students according to their mental models. Salleh et al. (2010) explored the effect of the personality trait of neuroticism on pair programming and reported that students' performance is not affected by different levels of neuroticism. Zacharis (2011) and Edwards, Stewart & Ferati (2010) investigated the effectiveness of online pair programming for introductory programming students. Zacharis found that students working online using pair programming produced code of better quality and more efficiently than students working individually. However, Edwards, Stewart & Ferati found that students were less satisfied with the experience of online pair programming than when co-located.

O'Grady (2012) reviewed the literature on the use of problem-based learning (PBL). More than a third of the 59 cases reviewed were first-year computing courses, and more than half of these were programming courses. O'Grady found that both teachers and students were largely positive about their PBL experiences. However, he found that the adoption of PBL into computing courses was largely ad hoc and random and concluded that if it is to be successfully used then "motivations, objectives, learning outcomes, and graduate outcomes must be clearly defined" (p 10). Sancho-Thomas, Fuentes-Fernández & Fernández-Manjón (2009) present the NUCLEO e-learning framework, a PBL-based environment for teaching computing courses. From the results of three different studies on the use of this framework the authors conclude that NUCLEO had a positive influence in decreasing dropout rates, raising exam pass rates, and improving team formation.

Social media and learning communities

Recently, various forms of social media (web 2.0) have been used in education programs to encourage collaborative work and the formation of learning communities. Using social media is also seen as a way to engage students in learning. A number of the implementations of contributing student pedagogy involve the use of social media (Hamer et al., 2011).

Pieterse and van Rooyen (2011) report the use of Facebook in a large first-year computer science unit. A closed Facebook group was set up as an informal online discussion forum complementing a formal discussion forum set up on the department website. Analysis of the usage of the forums showed greater use of the formal forum; however, there was more evidence of an online community on the Facebook forum. The authors' impressionistic view was that students were more engaged than in previous offerings of the course.

Two studies investigated the use of blogs to support learning communities. McDermott, Brindley & Eccleston (2010) describe the use of blogs in a collaborative and professional skills unit of a first-year computing course. Students were required to use a blog for a reflective diary and to post comments on other students' blog postings. The authors report that most students used their blogs in an educationally constructive way and the postings gave valuable insights into the students' experiences. Robertson (2011) describes the use of blogs in an introductory interactive systems course. Students were required to keep a design diary as a blog and to comment on the blogs of other group members. Analysis of the blogs gave insights into students' self-directed learning strategies and the support they provided to peers.

At an Australian university, Terrell, Richardson & Hamilton (2011) required students to record their reflections and learning activities on a blog. Analysis of the blogs provided indications as to how well the course objectives had been met. At another Australian university, Guo and Stevens (2011) used wikis for collaborative assignment work in an introductory information systems course. From the results of a student survey they provide recommendations for instructors who are considering using web 2.0 technology in their teaching programs.

Summary

There is a significant body of literature devoted to the theories and models of learning, various approaches to teaching, cooperative and collaborative learning techniques, and the use of social media. These were frequently discussed in terms of influences on student learning, motivation, and engagement.

A large proportion of this material was highly focused on the programming domain and only a small portion related specifically to the Australian context.

3.2 Current Practice in Australia

The interviews of Australian academics sought information about teaching practices in first-year ICT courses. The responses gave insights into current teaching practices and issues faced by teaching staff. Thematic analysis was used to extract and code the responses and to identify and define the major issues raised. The responses are discussed below under the main topics that were identified from the analysis of the interview data: "approaches to teaching", "cooperative and collaborative learning" and "social media and learning communities". These broadly align with three of the four topics from the literature search. An underlying theme across all topics is the response of academics to the perceived lack of student engagement with traditional methods of on-campus course delivery in universities, in particular the traditional lecture model of content delivery.

Approaches to teaching

A common element in this topic was the aim of increasing learner engagement through converting the learning experience from a passive activity of absorbing information to an active process whereby the learner must engage and process the content in order to construct meaning from the experience. The most dominant concerns regarding teaching were the issues involving lecture delivery and responses to the lack of student engagement with learning in this space.

Several interviewees raised the issue of lack of student attendance at lectures, and were making attempts to address this. For example, interviewee U7b indicated with regard to their lectures:

"Deliberate change to improve engagement. ... A complete change of staff, a complete change of pedagogy, a restructure of the delivery approach, etc. ... Because we found that the engagement and therefore the attendance and the interest ... is dropping off with this sort of generation. We've made a conscious decision to put our brightest performers, you might say, on first-year units."

In another example interviewee U15b discussed the rationale for the introduction of clicker technology into several first-year units:

"The other thing that is impacting the first-years is the use of clicker technology, ... And that has been in part to try and improve the lecture experience and also get attendance back up. You know that lecture attendance is the first thing that kind of goes when students are under pressure so we try to be quite compelling in having them

in there and them knowing why it is important and what they can get from it."

The consensus of comments indicated that it was important to provide students with an engaging and active lecture experience in order to motivate them to attend and participate in learning.

Lecture approaches that focus on transmitting content were seen as problematic since other sources of high quality information were available online in formats that could be accessed more conveniently off campus. A number of high quality MOOCs have focused on computing and ICT and are an example of the increased availability of resources of this type. The strengths of on-campus delivery were seen as being the ability to encourage active student participation, the responsiveness of lecturers in providing quality student feedback on progress, the social learning context involving their peers, and personalised feedback to students. Recently, lecture techniques and pedagogies have been developing to take advantage of these strengths.

One example of this process is the technique of the flipped classroom (Porter, Bailey-Lee & Simon, 2013; Simon et al., 2013) incorporating the use of clicker technologies. Interviewees U15a and U15b described the use of flipped classroom techniques and clicker technology specifically targeted at first-year students:

"Clickers were implemented...Pre-reading is expected. The way those lectures work is that there will be a quick summary and then there will be some sort of question posed to the class, they tend to discuss it in small groups, ... Students will get into small groups to discuss it and then they re-answer and then you can get a sense for how their understanding is shifting through a bit of discussion and prompting."

The aim of these techniques is to get students to actively engage with the fundamental concepts through a process of discussion and responses undertaken in conjunction with their peers. This also allows the lecturer to better judge the current state of understanding demonstrated by the class through their electronically submitted responses.

Interviewees U15a and U15b went on to indicate that the Faculty involved intended to expand the flipped classroom and clickers program further:

"What we found, which was actually quite good, is that it brought the tail up a bit. So we thought it might have a bit of an impact on students at risk ... " (U15b)

"It encourages them to actually attend. We're starting to have more units using clickers this semester." (U15a)

However, other interviewees indicated that they had implemented some components of the flipped classroom model but that it had proved problematic to motivate students to do the required pre-reading, so the approach was discarded. Further research is required on the impact of these techniques and technologies in the ICT domain and in the Australian context.

A variety of other approaches are used in lectures to engage students in learning experiences. Interviewee U24 uses live code writing and demonstrations to increase the interactivity of lectures. Interviewee U12 uses online quizzes within Moodle:

"Students can either use their phone, their computer or the tablet I provide to ensure that everyone has access. It's an online quiz so they get instant feedback as to how they've gone and I get the individualised feedback so I know who's struggling".

Role-playing is a novel approach used by two lecturers. Interviewee U23 explains:

"I do a lot of role play in lectures to try to reinforce some of the concepts. So I have people acting out variables and loops and things like that. It's a bit of a giggle, but students who struggle initially to try to understand these concepts seem to find that really helps".

Interviewee U23 shared his experience on having guest lectures in his course:

"We have guest lecturers every second week in the subject and try to mix them up across different fields so you get very engaging, inspiring people. ... We're very selective about who we approach to do [the lecture] and students love it. Of course we make that examinable so they actually have to come along to the guest lectures."

Despite many efforts to improve the lecture experience, some interviewees expressed strong negative views about it. Interviewee U5 encapsulates these ideas:

"I think the future of the lecture is in significant danger... students get very little value from lectures. The attendance is poor, the interaction is virtually all one way and today's students really don't see it as any benefit whatsoever... and the students are far busier now than they were 20 years ago when university may have been a priority. University isn't a priority anymore. The majority of our domestic students are working at least 20 hours a week and they see uni having to fit around them, not the other way round. I understand the challenges and there does have to be a nice balance but the changes have been quite dramatic and the universities are still teaching to the students as they were 30 years ago when students would come to class."

Although discussion of how teaching is approached was focused on the lecture environment during the interviews, a variety of other teaching techniques were mentioned that were appropriate for tutorial classes or online learning, often involving the use of specific tools and technologies. The motivation for these was to engage students in interesting and meaningful experiences.

Interviewee U9 explains how she focuses on students' interest to increase engagement:

"Every single week we have two or three 3-minute oral presentations by students on any topic of interest to them. Other students give feedback, because we're scaffolding their learning about how to present at the end of the semester. And that's great fun. .... They don't get marked on it; it' s formative".

Interviewee U6 argues that project work needs to be authentic to promote student engagement:

"The students engage in projects that are fascinating and do authentic tasks of real world challenges and coming up and creating something new. Not just learning by rote."

Similarly, interviewee U20 stresses the importance of providing opportunities to do meaningful and motivating work in his programming unit.

Interviewee U7b discusses the use of visual programming techniques based on a Stanford University model in which students learn to program by moving objects around a screen in a game-like environment in which the effects of the code and its successful execution are immediately apparent to the novice programmer.

"The ladybug is very visual. The aim is to run the code and see the ladybug move in the correct way instead of the old way of running the code and not getting an error and maybe producing a report. What you are seeing is a visual representation of your result. Quite a bit different to the old pedagogy."

Interviewee U10 describes the media computation introductory programming technique where students learn to program using the manipulation of images and sounds as the context for learning about programming.

"Media computation [is] really new. Introduced three years ago, [as a] first course for people who do not know anything about computing. People learn to program by manipulating images and sounds." Part of the rationale for this change was wider audience appeal, including for non-ICT students. So far, results have been positive.

"The students do seem to be more engaged, they are more enthusiastic, they are attending more classes, so we are taking that as a win enough at the moment."

Again there is a sense that there is not really an improvement at the higher end of student performance but more engagement at the lower end, with a possible consequence that more students are able to pass the introductory programming unit.

There were several comments in the interviews regarding the creation and use of educational resources. Interviewee U7a described an open educational resources (OER) scheme. This was a learning object repository of submitted student work that was created and maintained on a formal basis.

"Previous students' work can be referenced, can be extended, can be reused, and can be enhanced. That means the currently enrolled students can make use of previous students' work for improvements, for extensions and for some other kinds of extra work; however, students need to follow the OER scheme."

The aim was to build up a rich repository of student-generated content, and participation was voluntary.

Another interviewee, U15a, described an e-publishing initiative called Alexandria, based on WordPress infrastructure. The aim was to create dynamic and interactive learning objects that could be distributed on a variety of platforms. This is a type of e-publishing with interactive elements embedded, such as quizzes, applets and discussion forums.

"We have another project taking [the] online learning repository type thing and creating kind of learning modules. Again trying to do them in a more dynamic way, so short videos with interactive applets students can experiment with and stuff."

Cooperative and collaborative teaching

This topic is concerned with teaching approaches that involve students in collaborative or cooperative learning activities. Cooperative and collaborative learning activities were highlighted in the interviews as examples of active learning pedagogies for first-year students. Interviewee U10 explains:

"We do a lot of student contribution work in first year. ... it is very much based upon peer assessment and peer review, peers working together in collaboration. Our curriculum was restructured about 4 years ago now. We completely rebuilt the first-year curriculum around collaborative learning."

The aim here is to recast learning from being an isolated and solitary activity to being an intensely social activity where students are engaged and motivated by negotiating shared goals, responsibilities, and cooperative tasks involving their peers. The social nature of this learning experience and the intense engagement is intended to reduce the social isolation of students, which has been shown to be one of the significant risk factors for students dropping out of courses. Interviewee U10 elaborates: *"In the collaborative workshop sessions students do a lot of very active learning, they have little mini-lectures, that are interjected between collaborative learning activities where the students are often asked to build upon each others' work, to share each others' work and do peer review and peer assessment."*

Here the aim is to foster a range of skills related to the ability to plan solutions, negotiate roles, and evaluate progress, rather than just to absorb specific information. These social skills are deemed to be important in the context of future employment in the ICT field and tend to produce a more engaging learning experience.

According to interviewee U10, however, these collaborative learning techniques require a range of specific teaching techniques in order to ensure their successful implementation.

"They are very heavily guided through the workshops ... all face-to-face, so we have quite a lot of workshop supervisors who work with the groups. So the workshop supervisors go through training every year to sort of guide them into how to work with the student groups."

Further research is needed to formally describe and evaluate the impact of these techniques in the Australian ICT context.

A related active learning pedagogy is focused on problem-solving skills and in setting the frame of reference for learning activities in authentic problem contexts relevant to the ICT domain. Interviewee U9 provides an example:

"We have got peer collaboration within classes and some topics use partnership learning. And there is a student focus of what is going to be taught. There is a topic in which students undertake an external challenge of a real-world scenario for Engineers without borders ... our Computer Science, Engineering, and our ICT students participate in that, where they design real-world solutions for ICT problems in third-world countries. They design their own solution and it is incredible what they do in first year."

The innovation in this example is that this experience is targeted at first-year students in a professional skills unit rather than being delivered in a capstone unit in the third year. Students are motivated to gain skills as they go to complete the current project, rather than completing a series of units to gain a set of decontextualised prerequisite skills to be used at a later time.

Social media and learning communities

This topic is concerned with use of social media for learning activities in first-year ICT. Interviewee U24 describes the use of social networking software UCROO to support learning communities. UCROO is a social networking application for Australian universities only, and was developed by post-graduate students from Deakin University (which is not the university of interviewee U24). It is an educational social networking site based on Facebook.

"Looks a lot like Facebook, acts a lot like Facebook. The students are very familiar with it. They know how to use it immediately. It is unit specific, so you set a unit up in this. It definitely has an educational focus because you can set up assessment dates and the like. Each unit has a wall on which you can post, do a poll, ask questions, put up a file, link to a web page. But students can, too, so you get connections like you get Facebook friends. Everyone who is your friend, you have one common wall that you can see."

UCROO has a rich tool set of features to promote social connections and to allow posting of news and resources. This is very different from the limited tool set available in the current generation of LMSs. According to interviewee U24 the software was:

"Introduced 18 months, 2 years ago, to the introductory programming class, because they of course are a really quiet class because they are programmers, they tend to be quiet. They tend to be not so out there socially, and I also wanted my external students to get to know my internal students and for my internal students to be reminded that the class does not only consist of them."

The initial results have been positive:

"It has been magnificent, students have loved it and I have had an enormous amount of student interactivity as in [...] between students on UCROO each time I have used it. ... it actually really surprised me how these people just took to it like ducks would take to water."

The lecturer is also starting to build social networking tools more broadly into the unit, such as Skype for external presentations and web-based clicker systems for in-class polling.

However, several interviewees cautioned against the use of social media. As U4 explains:

"It is difficult to encourage students to use it because they think this is just another burden on what they're required to do."

Interviewee U7b remarked:

"The university is moving towards more social media but I think there are a few issues in using that extensively in teaching because students don't really distinguish between whether the social media contact is social or educational. It kind of blurs the boundaries for them."

The use of social networking has shown the potential to increase peer feedback, and to integrate online and on-campus students if implemented correctly. Further research and evaluation is required on the impact of social networking techniques on the ICT domain.

4 Discussion

Our analysis of recent literature shows that while there is a significant body of literature devoted to teaching in the first year of ICT courses, much of this literature is focused in the programming context. We propose that further research is needed to explore other aspects of the first-year ICT curriculum to gain a better understanding of the first-year ICT student experience.

The topics that emerged from an analysis of the interview data broadly align with those found in the literature. Most interviewees highlighted rapid changes in traditional methods of on-campus course delivery due to a perceived lack of student engagement, in particular changes to the lecture format and to the balance between lectures and practical labs. Practices such as active learning approaches, flipped classrooms, peer, cooperative, and collaborative techniques, and problem-based learning were frequently discussed, along with the integration of social networking tools to support the formation of learning communities. Again, the focus was predominantly on the programming context, so we propose that other areas of the first-year curriculum and the integration of the curriculum of the whole first year merit future consideration.

Finally there is a need to formally evaluate the effects of many of the innovative teaching practices described in this paper. Substantial work has been documented on efforts to improve the relevance and appeal of the ICT curriculum to a wider range of students, including non-ICT students, using social media, visual programming, and problem-based learning techniques. In many cases the initial reports of the techniques are positive, but more rigorous evaluation is required to support evidence-based decision-making on which techniques should be further developed to drive improvements in the first-year learning experience of ICT students.

5 Conclusion and future work

From this study we have documented a number of initiatives aimed at increasing ICT student engagement in the learning process. The study raises a number of key research areas that need further investigation. There is a clear need for more formal evaluations of the effects of these teaching initiatives in the Australian ICT context and for the collation of examples of good practice for wider dissemination. While initial results in many cases are positive, more evidence is required to justify sector-wide change. The amount of published literature on programming education also highlights a need to conduct research in other areas of ICT curriculum, to ensure a better overall first-year experience for ICT students.

6 Acknowledgements

This project was undertaken with the support of the Australian Council of Deans of Information and Communication Technology through the ALTA Good Practice Reports Commissioned for 2013–2014 grant scheme (http://www.acdict.edu.au/ALTA.htm).

The project team would like to acknowledge the work of Dr Beth Cook who worked as a research assistant to conduct the interviews and to prepare the detailed interview notes.

7 References

Anderson, M., & Gavan, C. (2012). Engaging undergraduate programming students: experiences using LEGO Mindstorms NXT. *13th Conference on Information Technology Education*, 139-144.

Apiola, M., Lattu, M., & Pasanen, T. A. (2010). Creativity and intrinsic motivation in computer science education : experimenting with robots. *15th*

Conference on Innovation and Technology in Computer Science Education, 199-203.

Apiola, M., Lattu, M., & Pasanen, T. A. (2012). Creativity-supporting learning environment – CSLE. *ACM Transactions on Computing Education*, 12(3), 11.

Bayzick, J., Askins, B., Kalafut, S., & Spear, M. (2013). Reading mobile games throughout the curriculum. *44th ACM Technical Symposium on Computer Science Education*, 209-214.

Beck, L., & Chizhik, A. (2013). Cooperative learning instructional methods for CS1 : design, implementation, and evaluation. *ACM Transactions on Computing Education*, 13(3), 10.

Biggs, J. (1996). Enhancing teaching through constructive alignment, *Higher Education,* 32(3), 347-364.

Cain, A., & Woodward, C. J. (2012). Toward constructive alignment with portfolio assessment for introductory programming. *IEEE International Conference on Teaching, Assessment, and Learning for Engineering 2012*, H1B-11.

Caspersen, M. E., & Kolling, M. (2009). STREAM: A first programming process. *ACM Transactions on Computing Education*, *9*(1), 4.

Collis, B. and Moonen, J. (2005). An On-Going Journey: Technology as a Learning Workbench, University of Twente, Enschede, The Netherlands.

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. *14th Australasian Computing Education Conference*, 77-86.

Corney, M., Teague, D., & Thomas, R. N. (2010). Engaging students in programming. *12th Australasian Computing Education Conference*, 63-72.

Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., & Saffrey, P. (2010). Manipulating mindset to positively influence introductory programming performance. *41st ACM Technical Symposium on Computer Science Education*, 431-435.

Daniels, T. E. (2009). Integrating engagement and first year problem solving using game controller technology. *39th IEEE Frontiers in Education Conference, 2009*, 1-6.

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. *43rd ACM Technical Symposium on Computer Science Education*, 141-146.

Dweck, C. S. (2000). *Self-theories: Their role in motivation, personality, and development*. Psychology Press.

Eagle, M., & Barnes, T. (2009). Evaluation of a game-based lab assignment. *4th International Conference on Foundations of Digital Games*, 64-70.

Edwards, R. L., Stewart, J. K., & Ferati, M. (2010). Assessing the effectiveness of distributed pair programming for an online informatics curriculum. *ACM Inroads*, 1(1), 48-54.

Falkner, K., & Palmer, E. (2009). Developing authentic problem solving skills in introductory computing classes. *ACM SIGCSE Bulletin*, 41(1), 4-8.

Guo, Z., & Stevens, K. J. (2011). Factors influencing perceived usefulness of wikis for group collaborative learning by first year students. *Australasian Journal of Educational Technology*, 27(2), 221-242.

Hamer, J., Luxton-Reilly, A., Purchase, H. C., & Sheard, J. (2011). Tools for contributing student learning. *ACM Inroads*, 2(2), 78-91.

Hamer, J., Sheard, J., Purchase, H., & Luxton-Reilly, A. (2012). Contributing student pedagogy. *Computer Science Education*, 22(4), 315-318.

Hanks, B., Murphy, L., Simon, B., McCauley, R., & Zander, C. (2009). CS1 students speak: advice for students by students. *ACM SIGCSE Bulletin*, 41(1), 19-23.

Heinsen Egan, M., & McDonald, C. (2014). Program visualization and explanation for novice C programmers. *16th Australasian Computing Education Conference*, 51-57.

Hertz, M., & Jump, M. (2013). Trace-based teaching in early programming courses. *44th ACM Technical Symposium on Computer Science Education*, 561-566.

Hu, M., Winikoff, M., & Cranefield, S. (2012). Teaching novice programming using goals and plans in a visual notation. *14th Australasian Computing Education Conference*, 43-52.

Hu, M., Winikoff, M., & Cranefield, S. (2013). A process for novice programming using goals and plans. *15th Conference on Innovation and Technology in Computer Science Education*, 3-12.

Hundhausen, C. D., Agrawal, A., & Agrawal, P. (2013). Talking about code: integrating pedagogical code reviews into early computing courses. *ACM Transactions on Computing Education*, 13(3), 14.

Kothiyal, A., Majumdar, R., Murthy, S., & Iyer, S. (2013). Effect of think-pair-share in a large CS1 class: 83% sustained engagement. *9th International Computing Education Research Conference*, 137-144.

Kurkovsky, S. (2013). Mobile game development: improving student engagement and motivation in introductory computing courses. *Computer Science Education*, 23(2), 138-157.

Lasserre, P., & Szostak, C. (2011). Effects of team-based learning on a CS1 course. *16th Conference on Innovation and Technology in Computer Science Education*, 133-137.

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. *13th Australasian Computing Education Conference*, 9-18.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1), 57-80.

Marsa-Maestre, I., De La Hoz, E., Gimenez-Guzman, J. M., & Lopez-Carmona, M. A. (2013). Design and evaluation of a learning environment to effectively provide network security skills. *Computers & Education*, 69, 225-236.

Mason, R., & Cooper, G. (2012). Why the bottom 10 % just can't do it – mental effort measures and implications for introductory programming courses. *14th Australasian Computing Education Conference*, 187-196.

McDermott, R., Brindley, G., & Eccleston, G. (2010). Developing tools to encourage reflection in first year students blogs. *15th Conference on Innovation and Technology in Computer Science Education*, 147-151.

McWhorter, W. I., & O'Connor, B. C. (2009). Do LEGO® Mindstorms® motivate students in CS1? *ACM SIGCSE Bulletin*, 41(1), 438-442.

Morazán, M. T. (2010). Functional video games in the CS1 classroom. *Trends in Functional Programming*, 166-183. Springer Berlin Heidelberg.

Murphy, C., Powell, R., Parton, K., & Cannon, A. (2011). Lessons learned from a PLTL-CS program. *42nd ACM Technical Symposium on Computer Science Education*, 207-212.

O'Grady, M. J. (2012). Practical problem-based learning in computing education. *ACM Transactions on Computing Education*, 12(3), 10.

Pears, A. (2010). Conveying conceptions of quality through instruction. *7th International Conference on the Quality of Information and Communications Technology*, 7-14.

Pears, A., & Rogalli, M. (2011). mJeliot: A tool for enhanced interactivity in programming instruction. *11th Koli Calling International Conference on Computing Education Research*, 16-22.

Pieterse, V., & van Rooyen, I. J. (2011). Student discussion forums: what is in it for them? *Computer Science Education Research Conference*, 59-70. Open Universiteit, Heerlen.

Porter, L., Bailey-Lee, C., & Simon, B. (2013). Halving fail rates using peer instruction: a study of four computer science courses. *44th ACM Technical Symposium on Computer Science Education*, 177-182.

Radermacher, A., Walia, G., & Rummelt, R. (2012). Improving student learning outcomes with pair programming. *9th International Computing Education Research Conference*, 87-92.

Risco, S., & Reye, J. (2012). Evaluation of an intelligent tutoring system used for teaching RAD in a database environment. *14th Australasian Computing Education Conference*, 131-140).

Robertson, J. (2011). The educational affordances of blogs for self-directed learning. *Computers & Education*, 57 (2), 1628-1644.

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

Salleh, N., Mendes, E., Grundy, J., & Burch, G. S. J. (2010). The effects of neuroticism on pair programming: an empirical study in the higher education context. *2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 22.

Salomon, G., & Perkins, D. (1988). Teaching for transfer. *Educational leadership*, 46(1), 22-32.

Sancho-Thomas, P., Fuentes-Fernández, R., & Fernández-Manjón, B. (2009). Learning teamwork skills in university programming courses. *Computers & Education*, *53*, 517-531.

Sheard, J., Carbone, A., & Hurst, A. J. (2010). Student engagement in first year of an ICT degree: staff and student perceptions. *Computer Science Education*, 20(1), 1-16.

Simon, B., Esper, S., Porter, L., & Cutts, Q. (2013). Student experience in a student-centered peer instruction classroom. *9th International Computing Education Research Conference*, 129-136.

Simon, B., Kohanfars, M., Lee, J., Tamayo, K., & Cutts, Q. (2010). Experience report: peer instruction in introductory computing. *41st ACM Technical Symposium on Computer Science Education*, 341-345.

Skudder, B., & Luxton-Reilly, A. (2014). Worked examples in computer science. *16th Australasian Computing Education Conference*, 59-64.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2), 8.

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4), 15.

Summet, J., Kumar, D., Hara, K. O., Walker, D., Ni, L., Blank, D., & Balch, T. (2009). Personalizing CS1 with robots. *ACM SIGCSE Bulletin*, 41(1), 433-437.

Sweller, J. (1999). *Instructional Design in Technical Areas.* Melbourne, Australia, ACER Press.

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. *16th Australasian Computing Education Conference*, 41-50).

Terrell, J., Richardson, J., & Hamilton, M. (2011). Using web 2.0 to teach web 2.0: a case study in aligning teaching, learning and assessment with professional practice. *Australasian Journal of Educational Technology*, 27(5), 846-862.

Thota, N., & Whitfield, R. (2010). Holistic approach to learning and teaching introductory object-oriented programming. *Computer Science Education*, 20(2), 103-127.

Wood, K., Parsons, D., Gasson, J., & Haden, P. (2013). It's never too early : pair programming in CS1. *15th Australasian Computing Education Conference*, 13-21.

Zacharis, N. Z. (2011). Measuring the effects of virtual pair programming in an introductory programming Java course. *IEEE Transactions on Education*, 54(1), 168-170.

Assessment in First-Year ICT Education in Australia: Research and Practice

Judy Sheard
Monash University
Australia
michael.morgan@monash.edu

Michael Morgan
Monash University
Australia
judy.sheard@monash.edu

Matthew Butler
Monash University
Australia
matthew.butler@monash.edu

Katrina Falkner
University of Adelaide
Australia
katrina.falkner@adelaide.edu.au

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Amali Weerasinghe
University of Adelaide
Australia
amali.weerasinghe@adelaide.edu.au

Abstract

This paper presents an investigation of assessment in first-year Information and Communications Technology (ICT) courses with a focus on Australian universities. This study was part of a project that aimed to identify and disseminate good practices in first-year ICT teaching in Australian universities. Through a systematic review of the last five years of research literature and interviewing 30 academics who were involved in the design and delivery of the first-year learning experience in Australian universities, we have formed a comprehensive view of current assessment practices, and outlined the unique challenges faced by teachers when designing assessment for their first-year ICT students. Key findings of the literature survey and the insights gained from the academic participants have been collated to provide examples of good practice in the field and to recommend areas for further investigation.

Keywords: First Year; Student Experience; Assessment; Academic Integrity.

1 Introduction

Assessment is a key component of the learning experience of university students. Assessment is used to measure the level of knowledge and skills that students have obtained, and determines their grades and course progression. Assessment can be used during the learning process to give students feedback on their work. An important consideration is that the form of assessment influences how students approach their study, with a consequent influence on learning outcomes (Biggs, 1996).

There are a variety of ways that students may be assessed, and the form of assessment used is often discipline-specific. For example, students learning to program may be assessed by a practical task on a computer. With recent moves to blended learning and technology-enhanced learning environments there are now new imperatives and opportunities for different forms of assessment.

Considering the central role of assessment in the student learning experience, it is critical that teachers choose the form of assessment that is appropriate for the learning situation and desired learning outcomes. In first-year courses it is also important to consider that students may not have encountered some forms of assessment in their previous education. The transition from secondary to tertiary studies is a difficult process for many students, first-year courses have high rates of attrition (Sheard, Carbone, & Hurst, 2010) and it is important to consider any possible influences on this experience.

In this paper we report findings of a study that investigated assessment practices in first-year Information and Communications Technology (ICT) courses in Australia. The study comprised a review of recent literature on assessment practices in ICT courses and a survey of Australian academics involved in teaching first-year ICT courses. The aims of the study were: 1) to gain a comprehensive view of how students in first-year ICT courses are assessed; 2) to determine factors influencing choice of assessment used; and 3) to identify examples of good practice in assessment in first-year ICT courses in Australia that could be adopted and disseminated widely. This study is part of a larger project exploring teaching practices in first-year ICT courses.

2 Research Approach

This section describes the approach used to investigate research and current practices in assessment in the first year of ICT courses in Australia. The investigation was conducted by the authors as part of a project that investigated the broader topic of research and practice in teaching ICT courses in Australia. To conduct the project, the team developed a framework with six themes that together describe the learning experience: ‘what we teach’, ‘where we teach’, ‘how we teach’, ‘how we assess’, ‘learning support’ and ‘student support’. As the focus of this paper is about assessment, only findings from the ‘how we assess’ theme will be reported.

Two phases were designed by the authors for this project; a systematic review of research literature from the previous five years, and interviews of academics involved in the delivery of first-year programs in

Australia. A detailed description of the methodology used in this project is reported in *Experiences of first-year students in ICT courses: good teaching practices*: Final Report: ICT student first year experiences (http://www.acdict.edu.au/ALTA.htm); accordingly, only a brief summary is presented below, focusing on the 'how we assess' theme.

Phase 1 of the project consisted of a systematic review of literature from 2009 to 2014 in the area of computing education. Keyword searches were carried out in Google Scholar and the IEEE Xplore and ACM Digital Library databases, along with manual searches of key computing education journals and conference proceedings.

In phase 2, semi-structured phone interviews were conducted with academics from Australian universities in February and March 2014. Participants were identified as key staff involved with the design and/or delivery of ICT courses to first-year students. Thirty academics from 25 Australian universities were interviewed. These included six Group of Eight (go8), three Australian Technology Network (ATN), six Innovative Research (IRU), and three Regional Universities Network (RUN) universities. The interviews averaged 53 minutes. Detailed notes were taken, and the interviews were audio recorded so that relevant comments could be transcribed at a later time. The interview script focused on six key themes, and all interviewees were sent the interview questions before being interviewed. Questions asked to elicit responses about initiatives in assessment practice included: 'What kinds of assessment items are used in the first-year courses?', 'For which assessment items is feedback given to students?', 'How much of the assessment is assessed automatically?', and 'For work not done in test conditions, what techniques are used to verify that the work is the student's own work?'. Follow-up questions on specific issues related to the themes were asked where appropriate.

3 How we assess

The investigation of assessment in first-year ICT courses in Australian universities covered the areas of assessment strategies, summative and formative forms of assessment, and tools to assess student learning or to facilitate the marking process. We begin our investigation of assessment in first-year ICT courses with a review of the literature. This gives a broad perspective of assessment in first-year ICT courses during the past five years, highlighting Australian studies. Following this, an analysis of the interviews of academics provides insights into assessment practices in Australian courses.

3.1 Literature Perspectives on Assessment in ICT

The systematic literature review found 38 papers that were concerned with assessment in university ICT courses during the previous five years. The literature on assessment was grouped into five topics:

- assessment design and strategies
- exam assessment
- non-exam assessment
- automated assessment
- assessment instruments and tools

All papers were set in the higher education sector. A high number of papers (27, 79%) dealt with assessment in first-year courses or assessment that was applicable to the first year. Most papers (33, 87%) dealt with issues concerning assessment of programming, and almost half (18, 47%) were Australian studies.

Assessment design and strategies

A couple of papers were found that focused on assessment of first-year students in university courses in general. A review by Yorke (2011) of assessment and feedback practices in the first year of university highlights the importance of early and timely feedback and a pedagogy that encourages students to reflect on their learning. A comprehensive report by O'Neill and Noonan (2011) presents a series of resources to assist in designing assessment tasks. An underlying principle is to build first-year students' confidence with low-stakes assessment before moving progressively to high-stakes assessment. Staff are encouraged to restrict the amount of assessment they build into their units to allow students time and opportunity for in-depth engagement with the teaching program. This strategy is based on the idea that to be successful in learning, students need to be engaged and empowered.

A number of papers deal specifically with assessment strategies in ICT courses. Taking a holistic view of the assessment process in programming courses, Australian researchers Thomas, Cordiner, and Corney (2010) propose the 'teaching and assessment of software development' framework (TASD) and give examples of its use across multiple year levels. Barros (2010) discusses the importance of assessment strategies in introductory programming and proposes a set of techniques and criteria to consider when designing programming assessment and grading. For assignment work he incorporates a plagiarism detection tool and oral assessment, and for the final practical exam, a minimum acceptable grade. Both papers report positive results in terms of student satisfaction and higher grades.

A problematic area for assessment in ICT courses is group work. An Australian researcher (Richards, 2009) discusses ways of assessing group work, including peer assessment, and the challenges of providing a fair distribution of marks to each group member. Hahn, Mentz, and Meyer (2009) investigated different forms of assessment for pair programming, and propose that a combination of self, peer, and facilitator assessment can increase the amount of feedback to the students, resulting in higher levels of achievement.

Exam assessment

A final written exam is a common form of summative assessment in computing courses. A number of papers reported studies of exam assessment, and these were all in the context of introductory programming. Much of this work has been conducted by Australian researchers.

Petersen, Craig, and Zingaro (2011) analysed 15 introductory programming exams to determine the types of question and the topics they covered. They concluded that some questions were too difficult for introductory programming students due to the high number of concepts students were required to understand in order to answer each question.

A corpus of work led by Australian researchers has investigated the use of formal examinations for the summative assessment of programming. The initial phase of this research investigated the structure of programming exam instruments, including an in-depth study of the types of question used. This involved development of a scheme to classify programming questions on a number of dimensions including style, course content, skill required to answer, difficulty, and complexity (Sheard et al, 2011). The classification scheme was applied to questions in 20 programming exam papers from multiple institutions (Simon et al, 2012). The study found that introductory programming examinations vary greatly in the coverage of topics, question styles, skill required to answer questions, and the level of difficulty of questions. Harland, D'Souza, and Hamilton (2013) used the same classification scheme to further explore question difficulty. The next phase extended this work to design a set of questions suitable for benchmarking in introductory programming courses (Sheard et al, 2014).

Another aspect of this work was an investigation of the pedagogical intentions of the educators who construct exam instruments (Sheard et al, 2013). This involved interviews with programming teachers to gain an understanding of how they go about the process of writing an exam, the design decisions they make, and the pedagogical foundations for these decisions. The study found that the process of setting exams relied largely on intuition and experience rather than explicit learning theories or models. Exam formats are typically recycled and questions are often reused. While there is variation in the approaches taken to writing exams, all of the academics take a fairly standard approach to preparing their students for the exam. Although some academics consider that written exams are not the best way to assess students, most tend to trust in the validity of their exams for summative assessment.

Another group of Australian researchers investigated summative assessment of introductory programming, focusing on the use of multiple-choice questions in exams (Shuhidan, Hamilton, and D'Souza, 2009; 2010). Most instructors in their study considered multiple-choice questions appropriate for testing questions on the lowest levels of the Bloom taxonomy (Bloom, 1956), but less than half were confident that multiple-choice questions could be used to test understanding of programming concepts (Shuhidan, Hamilton, and D'Souza, 2009; 2010). A problem faced in the investigation of exam questions is the difficulty in applying Bloom's taxonomy to classify exam questions according to their cognitive level. An Australian research team has developed an online tutorial to train researchers in the use of this and other taxonomies (Gluga et al, 2013).

Another Australian researcher (de Raadt, 2012) investigated the use of 'cheat sheets' in introductory programming exams and found that students who took permitted hand-written notes into their exam performed better than students who did not have notes.

Non-exam assessment

Research studies on forms of assessment other than examinations focused mainly on assessment of programming. Studies of both summative and formative assessment were found, with some reporting innovative practices.

A common form of in-semester assessment is the programming assignment. A grounded theory study by Kinnunen and Simon (2010; 2012) explored introductory programming students' experience of their assignments, and found that students' self-efficacy is not necessarily related to their experiences of success in programming.

A novel approach by Lee, Ko, and Kwan (2013) embedded assessment into an educational computer game designed to teach programming. A study of students' use of this game showed that incorporating assessment increased students' use of the game, the levels they achieved, and the speed at which they played the game.

Portfolio-based assessment is rather less common than assignments. Australian researchers Cain and Woodward (2012) describe an introductory programming unit where students are assessed entirely on a portfolio of work produced during the semester. The design of the unit was founded on Biggs's constructive alignment (Biggs, 1996), which proposes alignment between the learning activities, assessment, and intended learning outcomes. An evaluation showed that students were positive about their learning experience. Pears (2010) reports on the use of portfolio assessment in an introductory programming unit for the purpose of implementing a continuous assessment model. He found that students who completed the unit produced code of a higher quality than typically produced by first-year students.

Peer review is a form of assessment used for both formative and summative assessment. Assessing the work of peers can encourage student engagement and deeper learning (Carter et al, 2011). Peerwise, a collaborative web-based tool, enables students to create and share multiple-choice questions and allows students to peer-review questions submitted by others. Evaluation of the use of Peerwise has shown that it can foster student engagement and have a positive impact on learning (Denny, Hanks, and Simon, 2010; Purchase et al, 2010).

The use of social media (web 2.0) in education has led to new forms of assessment where students demonstrate their learning through online tasks that are often co-created and visible to their peers, and, in some cases, to wider audiences. These new forms have brought challenges for students and teachers in using unfamiliar authoring tools and applying appropriate citation and referencing to their work. Studies by Australian researchers Gray et al (2010) investigated examples of assessment using different web authoring tools and showed how principles of good assessment practice were reflected in each case. Further studies investigated the affordances of web 2.0 technologies for assessment, along with issues of ownership, privacy, and visibility of work (Gray et al, 2012; Waycott et al, 2013). A case study by Terrell, Richardson, and Hamilton (2011) describes assessment of a web 2.0 task in an introductory information management course under the framework of constructive alignment.

Automated assessment

The time-consuming tasks of collecting, marking, and giving feedback to students on their assessment work has led to the development of tools to help manage these processes. All of the assessment tools that we found were specifically designed for use in introductory programming classes.

Law, Lee, and Yu (2010) present PASS – Programing Assignment aSsessment System. PASS provides feedback for programming assignments by executing a set of instructor-prepared test cases and then comparing the expected output with the actual output. PASS also allows the teachers to monitor the testing process of students' submissions in real time and to share with the entire class examples that demonstrate good practice. A study of PASS showed a positive impact on students' self-efficacy.

Wang et al (2011) discuss the role of automatic assessment in introductory programming and present a tool, AutoLEP, for automatic analysis and assessment of student programs. They describe their use of this tool for in-semester formative assessment and for end-of-semester exams. Students and staff were enthusiastic about the tool, with staff reporting that students showed increased interest in programming and improvement of their skills.

Llana, Martin-Martin, and Pareja-Flores (2012) present an online free laboratory of programming (FLOP), which hosts a repository of programming problems that students can attempt and have automatically assessed. Preliminary results indicate positive improvement in students' motivation, skills, and self-efficacy.

Johnson (2012) presents a tool, SpecCheck, for testing conformance of programs to the assignment specification prior to submission. A small study showed that students were willing to accept having to produce highly structured homework in return for faster grades and feedback.

Shaffer and Rossen (2013) present the Programming Learning Evaluation and Assessment System for Education (PLEASE), a code-checking and submission system. Using data collected from the system, the lecturers were able to identify parts of the course where students were experiencing difficulties and make adjustments to the teaching program. The results of a small study indicated that the tool was useful in optimising course structure.

Assessment instruments

A few studies report the development of specialised assessment instruments. Ford and Venema (2010) trialled the use of short objective tests to test students' knowledge of fundamental programming concepts after their introductory programming course. Gouws, Bradshaw, and Wentworth (2013) designed a test to determine students' computational thinking ability prior to entering their computer science course. Elliott Tew and Guzdial (2010) propose a method for developing a language-independent assessment instrument for introductory programming.

The apparent prevalence of plagiarism and collusion is a topic of concern in the assessment of introductory programming. Australian researchers Nguyen et al (2013) present a source code similarity reporting tool developed as a Moodle plugin. Studies of staff and student reaction to the tool showed its usefulness in deterring and detecting plagiarism and its potential as an educative tool.

Summary

The literature on assessment in first-year ICT courses relates predominantly to programming. Nearly half of the papers found were from the Australian context, indicating research strength in this area. Although exam assessment has attracted the most research, a number of other forms of assessment have been investigated. Underlying motivations for academics' choice of assessment were often pedagogical: to encourage student engagement, provide timely feedback, or ensure academic integrity; or they were pragmatic: to ease the burden of marking. With the trend of an increased reliance by students on online course materials, further research is suggested on methods to improve the automation of assessment and provide quality feedback on students' work, while maintaining the academic integrity of the assessment process.

3.2 Current Assessment Practice in Australia

The interview questions sought information about assessment practices in first-year ICT courses in Australia. The responses gave insights into current assessment practices and issues faced by teaching staff. Thematic analysis was used to extract and code responses and to identify the major issues raised. The responses to these questions are discussed under the main topics that were identified from the analysis of the interview data: assessment design and strategies, exam and non-exam assessment, and automated assessment. The issues of provision of feedback, verification of student work, and other issues associated with academic integrity are discussed in terms of the different forms of summative and formative assessment. In reporting the findings, representative quotes have been included to further elucidate the discussion.

Assessment design and strategies

Students in first-year ICT courses are typically assessed via an end-of-semester written examination following in-semester tasks that may include assignments, portfolios, tests, tutorial exercises, or presentations. The most common assessment models used are assignment work and a final exam combined with either a mid-semester test or tutorial assessment.

A couple of interviewees mentioned their university having an overall assessment strategy. Interviewee U8 commented that at her university, "*assessment revolves around problem solving – looking at authentic situations*". An assessment guide based on Biggs's theory of constructive alignment (Biggs, 1996) had been developed at one university. Constructive alignment was also mentioned as a theoretical basis of portfolio assessment at another university.

A number of interviewees had designed assessment strategies to address the issue of lack of student engagement. Interviewee U7a explains:

"Previously, I have implemented some unit rules for encouraging student engagement. For example, the tutorial attendance is no lower than 85%. That will be recorded. Secondly, students' tutorial attendance is marked and also we have some in-class quizzes."

Most interviewees mentioned assessment policies at their university. It is common practice to set thresholds that students must reach in exams in order to pass a unit. Most often the threshold is 50%, but 40% is also used. Several interviewees mentioned mandated percentages of supervised work. In order to avoid over-assessment, some universities limit the number of assessment tasks per semester. In a couple of cases, a maximum of four assessment items was allowed; and in another case two major assignments and an exam were recommended. At one university it was a policy to provide feedback on an assessment task within 2 weeks, and to have an assessment task within the first 5-6 weeks of the semester in order to give early feedback to students.

Exam assessment

An end-of-semester written exam is the typical form of summative assessment in first-year ICT courses. Exams are seen as necessary to verify that it is the student's own work that is being assessed; however, some interviewees expressed concerns that a written exam is not necessarily a good method for establishing what the students have learned. One interviewee mentioned a move away from exams at her institution but not for first-year courses. Most exams are weighted between 40% and 60% of the overall mark for a unit, with 50% the most common weighting. The lowest weighting was 20% and the highest was 70% of the overall mark.

The use of multiple-choice questions in exams varies, and appears to be controversial. One interviewee sets most of the exam (and mid-semester test) as multiple-choice questions due to a large enrolment (250 students). Another uses multiple-choice questions in exams but says that more than 50% of assessment using multiple-choice questions would be frowned upon at his university. Interviewee U17 sets an exam of multiple-choice questions, arguing that: "*the only other option I can think of is to have programming problems on the exam paper but the exam is not the place where you can do any thinking.*"

Non-exam assessment

In combination with an end-of-semester exam there are a variety of other forms of summative assessment. The most common is assignment work, done individually or sometimes in a group. Often more than one assignment is set during the semester. Some interviewees mentioned checkpoints for assignments where students must show their tutor their progress. Checkpoints are incorporated to encourage students to start work early and to give them feedback. However, they are also used to monitor their work, which can help determine whether the student has done the work submitted.

Tests held during semester are a common form of assessment. These may be mid-semester tests worth from 10% to 20% or a series of smaller tests often conducted online using the LMS or another tool, such as ViLLE (Rajala et al, 2007). Some interviewees expressed a preference for continuous assessment, with smaller tests rather than one larger test. One interviewee commented that he does not hold a mid-semester test as the semester is only 11 weeks long.

Another common form of assessment is tutorial work. This involves assessment of tasks performed in the tutorial, often on a weekly or fortnightly basis. Typically this is low-stakes assessment with a few marks (1-2%) allocated for each assessment item. Interviewees mentioned that assessment in tutorials is a strategy for encouraging students to come to class and to work in class. An additional benefit was that tutors could observe students working and alert them to possible cases of plagiarism. However, interviewee U18, while acknowledging the benefits of lab assessment, found that it was "*more trouble than it was worth*".

Some universities use portfolio assessment. At one university portfolio assessment is embedded into each year level, and students are given training in their first year to help them understand the expectations of this form of assessment.

At another university portfolio assessment has been used for the past five years in an introductory programming unit. The portfolio assessment has been designed using Biggs's constructive alignment. Interviewee U1 explains:

"*This has been one of the changes that I think had a big impact as well on the pass rates for the introductory programming unit … a large change, moving away from assignments and exams to submitting a portfolio of assessments.*"

Interviewee U1 describes the process:

"*Each week the student will develop pieces of work that demonstrate how they've met one or all of the unit learning outcomes and each week we have a formative feedback process. With the portfolio assessment it has weekly feedback. It's 100% portfolio assessed so they don't get a grade until the end of the semester.*"

Interviewee U1 goes on to explain the grading process at the end of semester:

"*Each student has to submit a portfolio that demonstrates how they have met all of the unit learning outcomes. Then there is a scale by which they can meet [the learning objectives]. To meet them to an adequate level there are criteria. To meet them to a credit level there are separate criteria, and so on for distinction and high distinction. This allows students to work to their expectations. Some students only want to pass the unit and they're not interested in doing really well … That's not what their goal is in life.*"

At this university the portfolio assessment was a big change in the way the introductory programming is taught and students are assessed:

"*Each week the students submit work to get feedback so that they can improve that work and thereby improve their understanding. There's no punishment for doing that. Previously if students did an assessment at the beginning of semester and did poorly they lost those marks and they can never get them back. ... With this what we can do is go back and really focus on those very first things they didn't understand and make sure they understand those before they move on to the next thing. Some people might take a few weeks to get through the first few tasks they have to complete whereas others might get them done very quickly.*"

Other less common forms of assessment mentioned were presentations and submitted homework tasks; one interviewee gave students a mark if they visited the lecturer to ask a question.

There were indications of a growing use of social media for assessment tasks. For example, interviewee

U7a allowed students to use social media to deliver an e-learning information resource that they developed as an assignment task. Interviewee U24 discussed how he uses blogs and UCROO, an educational social-networking site based on Facebook. However, another interviewee raised a concern related to plagiarism when using social media: "*We've told them not to talk about the assignment but it's hard to police so I discourage it because of the plagiarism issue.*"

Automated assessment

Automated assessment is not used to any great extent in most universities. The most common use is for quizzes and multiple-choice question components of tests and exams. There were some examples of automatic testing of programming assignments. Interviewee U18 said that automatic assessment was used for: "*80% of the marks – none of it is automatic, but all of it has automated support.*"

Feedback

The comments by interviewees indicate that feedback is an important part of the assessment process. At most institutions feedback is given on all forms of in-semester assessment. Formative feedback on assignments is often given verbally during tutorials or consultation times. Portfolio assessment allows for continuous formative feedback throughout the semester. Feedback on summative assessment is typically given verbally for tutorial tasks and is written on assignment work. In the case of class tests, feedback is usually just a score.

A number of interviewees described providing detailed critiques for summative assessment of assignment work involving comments and scores for individual components. Assignment work is often assessed using rubrics. A couple of interviewees stated that they give feedback on assignment work as a summary at a lecture. In one case feedback on assignment work is given only in this open forum; however, students are also given the opportunity to discuss their work individually with their lecturer.

Some interviewees mentioned particular approaches to giving feedback for assignments submitted online. The GradeMark tool from Turnitin was mentioned by some as facilitating provision of feedback through dragging and dropping of comments. Interviewee U9 details a university-wide policy of e-assessment:

"*All student work must be submitted online and returned online, and that was trialled last year and has gone live this year. So we have been embedding feedback in online assessment.*"

At interviewee U9's university all assignment submission times are recorded and therefore the timeliness of the feedback provided to students is also recorded. A permanent record of all feedback is also stored, in case an issue arises. This university-mandated policy has the potential effect of allowing an audit of the quality and promptness of the feedback provided to all students in every course. Therefore a systematic process may be implemented to improve the standard and responsiveness of the feedback delivered to students.

Some assessment tasks enable instant feedback on performance. Examples are online quizzes and programming assignments with automated assessment. One interviewee commented that the instant feedback was very popular with the students.

The only feedback on exams is through viewing the exam script. Most interviewees indicated that very few students do this. Interviewee U16 stated that at his university comments are written on the exam scripts with the expectation that at least some students will come and look at them.

Academic integrity

Three subthemes emerged from analysis of the academic integrity theme.

Verification of work

In trying to determine whether a submitted assessment task is the work of the student submitting it, the interviewees use a range of strategies including interviewing, monitoring and observing.

Most agreed that interviewing students about their submitted assignment work was an effective way of verifying that the work was their own and identifying possible cases of plagiarism or collusion. A couple of interviewees described thorough interview processes. For example, interviewee U18 commented "*At the interview they are expected to discuss the code they've written and make changes to it.*" Interviewee U15b proposed that an interview does not have to be long to be effective:

"*You can [ask] just a few pointed questions about their motivation for the design they made, why they did it that way, and you can start to poke them a bit and say 'if we change this what would happen?'; 'if you wanted to do this feature how would you do it?'. I've used the interview and they tend to be pretty good at picking up where it might not be all the student's own work.*"

Despite its acknowledged effectiveness, interviewing every student as part of the assessment process is used in only a few institutions, typically in programming units. Many interviewees claimed that they have too many students and too few resources to conduct interviews. Interviewing had recently been abandoned at a couple of universities. As interviewee U16 explained, interviewing was "*extremely effective but very time-consuming, so we just couldn't keep it up*." A number of interviewees said that they interviewed students only if they were suspicious of the work. Interviewee U12 said that interviews are not used in her university because the previous head of school was concerned that "*it could mean asking different questions of different students and could cause [equity] issues.*"

Sometimes there are opportunities for less formal verification approaches where students can be questioned in their tutorials during the formative stages of an assignment. Some interviewees are alerted to possible cases of plagiarism through monitoring students' work and observing patterns of participation. Interviewee U24 incorporates a tutorial participation mark as part of the assignment mark, stating that: "*it's actually a way of encouraging students to work every week and it's also a way of controlling plagiarism.*"

Tools are sometimes used in verification of student work. The plagiarism detection tool Turnitin is frequently used for text-based assignments; however, the use of plagiarism detection tools for programming assignments appears less common. Tools such as MOSS (Measure of

Software Similarity), JPlag, and ESP were mentioned for detection of code plagiarism; however, one interviewee suggested that plagiarism detection tools were not suitable for first-year programming as there is usually too much similar code. Interviewee U2 only follows up on obvious plagiarism, seeing the assignments "*as learning opportunities as much as assessment.*"

However, plagiarism detection tools are not useful in detecting cases where students have commissioned their assignment work. Some interviewees rely on the assignment markers noticing disparities either within the submitted work or between the submitted work and the student's normal work. As interviewee U6 explained:

"*you get a pretty good eye for it once you've marked a few things and you know the standard or the hallmark of the student's work and if something significantly deviates from that you can start looking into that. I'll always keep an eye out for phrases or chunks of text that look like they've been written in a different style.*"

However, this becomes more difficult in large classes with multiple markers, and does not always cover the cases where someone else has done the work. A couple of interviewees mentioned that they had found their assignments advertised on a code-purchasing site. A strategy used by interviewee U22 is to give each assignment a unique name to make it easy to do a Google search to find any plagiarised code. Another interviewee mentioned a network of universities that monitored code-purchasing sites to pick up on cases where assignments had been commissioned.

Discouraging cheating

A number of strategies were used to discourage cheating. All universities had invigilated assessment in at least the exam component. As interviewee U20 noted, "*the only thing you can absolutely guarantee are the moderated parts, which are the exams.*" In a number of universities, students were required to gain a minimum exam mark, typically 40% or 50%, to pass a unit. A couple of interviewees commented that they used exams to pick up on students who had not done their own assignment work. However, Interviewee U4 noted that his university has a policy that "*exams are not to be for the purpose of ensuring that people haven't cheated.*"

Interviewees suggested a number of strategies to encourage students to do their assignment work. These were seen as preferable to punitive approaches. Some stress to their students that writing code on their own will help them with their exam. One interviewee uses careful assessment design where assignments are not just taken from the textbook; a couple of others set assignments tailored to individual students, allowing students to negotiate their own assignment. There was no consensus about whether students should work individually or with others on their assignments. Interviewee U19 permits students to work their assignments in pairs as he considered that "*this makes it much less likely that they will seek outside help*"; whereas at another university all first-year assignments are individual.

Two interviewees explained how they use email messages to discourage plagiarism, either sent from the lecturer …

"*I would send an email to students normally around that the time the assignment is due because I think most plagiarism occurs when students get behind and the assignment is due and they quickly find a friend to copy from. I tell them that if they have fallen behind to ask me, not their mate.*" (U13)

… or sent from the head of the school every semester:

"*…every semester the HoS sends an email to all students saying there were X number of students found guilty of plagiarism this semester and you should all be taking this seriously. So he also gives feedback to students about what students have been caught plagiarising to show them that we're actually catching them and doing something about it.*" (U17)

Two interviewees also mentioned how Turnitin is used to discourage plagiarism through detection. Interviewee U25 mentioned: "*We advise the students that their assignments would be put through Turnitin*" and interviewee U5 mentioned: "*They're all very well aware of Turnitin because when they put their assignment in they get a report back.*"

Penalties for breaches of academic integrity

Every university has a standard procedure to deal with academic breaches. Most universities have a designated officer to ensure that standard penalties are imposed across the school, the faculty, or the university. Substantial breaches are dealt with at the higher levels of management outside the particular school. For example, a dean's review was required to deal with substantial breaches in one university. Many universities maintain details of academic breaches in a central register or in the individual student's file.

The penalties imposed depend on the severity of the breach, the weighting of the assignment, and whether it is a repeat offence. Penalties range from zero marks for the specific assessment, to failing the unit, all the way through to being excluded from the university. Interviewee U23 said that for repeat offenders "*it could go all the way to a student having their enrolment terminated, which would be a very rare thing, but it has happened in the past.*"

Interviewee U12 discussed the importance of understanding the overall situation when an academic breach occurs:

"*However, it's not just 'OK, you've plagiarised, you're going to get this penalty'. It's looking at the circumstances around it and what's happened; whether they've understood what plagiarism is. And whether they've acknowledged what's happened.*"

When asked what would happen to a student who had copied something from the Internet and it was their first offence, interviewee U9 explained:

"*They would be educated and make sure that they do the quiz [students are expected to complete an academic integrity quiz which is 5% of their overall grade]. They would be told about proper paraphrasing and citing sources etc.*"

4 Discussion and Recommendations

Although a variety of forms of assessment were identified in the literature, most interviewees mainly discussed traditional forms of assessment. The few innovative assessment practices found were designed to encourage attendance (e.g. tutorial assessment), engage students in learning activities (e.g. social media), or encourage good work habits (e.g. portfolios). Interviewees' comments indicated the high importance they place on giving feedback on work during semester. Academic misconduct is a problematic area and there are a range of techniques used to verify students' work and discourage plagiarism and collusion.

A number of areas identified concerning assessment practice warrant further investigation. Overwhelmingly, the context for research and discussion in assessment was in the context of programming. There were a variety of techniques and tools for assessment of programming, but very few in other areas of study. We suggest that research on assessment techniques for other areas of the first-year ICT curriculum might be appropriate. The recent adoption of social media has led to innovative forms of assessment and there were reports of its use in a number of universities; however, few studies were found that evaluated the use of this assessment form in first-year ICT courses. This is an area that could be further investigated.

A key issue raised by interviewees was that the trend for increased online delivery had placed demands on academics to create appropriate assessment tasks for this context and to verify the identity of the student undertaking the assessment. There is a clear need for work in this area. Related to this, there was a perceived need for more tools to automate assessment and facilitate feedback for large groups. We propose that these issues require further research in order to ensure valid and fair assessment for our first-year students.

5 Conclusions and Future Work

Our investigation of assessment in the first year of ICT courses found that most of the literature is related to assessment in programming courses. Assessment of programming is an active area of research in Australia, although most of the work is focused on exam assessment. In contrast, the good practices in assessment identified in Australian ICT courses are concerned with portfolio assessment, interviewing students to verify assignment work, and using appropriate tools to facilitate and expedite provision of feedback for in-semester tasks and assignments.

Assessment is a key part of the total learning experience of our ICT students and has a major impact on their educational outcomes. This study contributes to our knowledge of assessment practices in first-year ICT courses and motivations and impediments to their use.

6 Acknowledgements

This project was undertaken with the support of the Australian Council of Deans of Information and Communication Technology through the ALTA Good Practice Reports Commissioned for 2013–2014 grant scheme (http://www.acdict.edu.au/ALTA.htm).

The project team acknowledges the work of Dr Beth Cook, who worked as a research assistant to conduct the interviews and to prepare the detailed interview notes.

7 References

Barros, J.P. (2010). Assessment and grading for CS1: towards a complete toolbox of criteria and techniques. *10th Koli Calling International Conference on Computing Education Research*, 106-111.

Biggs, J. (1996). Enhancing teaching through constructive alignment, *Higher Education*, 32(3), 347-364.

Bloom, B.S. (1956). *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company.

Cain, A., & Woodward, C.J. (2012). Toward constructive alignment with portfolio assessment for introductory programming. *IEEE International Conference on Teaching, Assessment, and Learning for Engineering 2012*, H1B-11.

Carter, J., Bouvier, D., Cardell-Oliver, R., Hamilton, M., Kurkovsky, S., Markham, S., McClung, O.W., McDermott, R., Riedesel, C., Shi, J., & White, S. (2011). ITiCSE 2010 working group report: motivating our top students. *16th Conference on Innovation and Technology in Computer Science Education – Working Group Reports*, 1-18.

Denny, P., Hanks, B., & Simon, B. (2010). Peerwise: replication study of a student-collaborative self-testing web service in a US setting. *41st ACM Technical Symposium on Computer Science Education*, 421-425.

de Raadt, M. (2012). Student created cheat-sheets in examinations: impact on student outcomes. *14th Australasian Computing Education Conference*, 71-76.

Ford, M., & Venema, S. (2010). Assessing the success of an introductory programming course. *Journal of Information Technology Education*, 9, 135-145.

Elliott Tew, A., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. *41st ACM Technical Symposium on Computer Science Education*, 97-101.

Gluga, R., Kay, J., Lister, R., Simon, & Kleitman, S. (2013). Mastering cognitive development theory in computer science education. *Computer Science Education*, 23(1), 24-57.

Gouws, L., Bradshaw, K., & Wentworth, P. (2013). First year student performance in a test for computational thinking. *South African Institute for Computer Scientists and Information Technologists Conference*, 271-277.

Gray, K., Thompson, C., Sheard, J., Clerehan, R., & Hamilton, M. (2010). Students as web 2.0 authors : implications for assessment design and conduct. *Australasian Journal of Educational Technology*, 26(1), 105-122.

Gray, K., Waycott, J., Clerehan, R., Hamilton, M., Richardson, J., Sheard, J., & Thompson, C. (2012). Worth it? Findings from a study of how academics assess students' web 2.0 activities. *Research in Learning Technology*, 20(1), 1-15.

Hahn, J.H., Mentz, E., & Meyer, L. (2009). Assessment strategies for pair programming. *Journal of Information Technology Education*, 8.

Harland, J., D'Souza, D., & Hamilton, M. (2013). A comparative analysis of results on programming exams. *15th Australasian Computing Education Conference*, 117-126.

Johnson, C. (2012). SpecCheck: automated generation of tests for interface conformance. *17th Conference on Innovation and Technology in Computer Science Education*, 186-191.

Kinnunen, P., & Simon, B. (2010). Experiencing programming assignments in CS1: the emotional toll. *6th International Computing Education Research Conference*, 77-86.

Kinnunen, P., & Simon, B. (2012). My program is ok – am I? Computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1), 1-28.

Law, K.M.Y., Lee, V.C.S., & Yu, Y.T. (2010). Learning motivation in e-learning facilitated computer programming courses. *Computers & Education*, 55(1), 218-228.

Lee, M.J., Ko, A.J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. *9th International Computing Education Research Conference*, 153-160.

Llana, L., Martin-Martin, E., & Pareja-Flores, C. (2012). FLOP, a free laboratory of programming. *12th Koli Calling International Conference on Computing Education Research*, 93-99.

Nguyen, T.T.L, Carbone, A., Sheard, J., & Schuhmacher, M. (2013). Integrating source code plagiarism into a virtual learning environment : benefits for students and staff. *15th Australasian Computing Education Conference*, 155-164.

O'Neill, G., & Noonan, E., (2011). *Designing First Year Assessment Strategically*, 1-46. http://www.ucd.ie/t4cms/designifyassess.pdf, accessed 24 Jun 2014.

Pears, A. (2010). Conveying conceptions of quality through instruction. *7th International Conference on the Quality of Information and Communications Technology*, 7-14.

Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing CS1 exam question content. *42nd ACM Technical Symposium on Computer Science Education*, 631-636.

Purchase, H., Hamer, J., Denny, P., & Luxton-Reilly, A. (2010). The quality of a PeerWise MCQ repository. *12th Australasian Computing Education Conference*, 137-146.

Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2007). VILLE: a language-independent program visualization tool. *Seventh Baltic Sea Conference on Computing Education Research*, 151-159.

Richards, D. (2009). Designing project-based courses with a focus on group formation and assessment. *ACM Transactions on Computing Education*, *9*(1), 2.

Shaffer, S.C. & Rossen, M.B. (2013). Increasing student success by modifying course delivery based on student submission data. *ACM Inroads*, 4(4), 81-86.

Sheard, J., Carbone, A., & Hurst, A. J. (2010). Student engagement in first year of an ICT degree: staff and student perceptions. *Computer Science Education*, 20(1), 1-16.

Sheard, J., Simon, Carbone, A., D'Souza, D., & Hamilton, M. (2013). Assessment of programming: pedagogical foundations of exams. *18th Conference on Innovation and Technology in Computer Science Education*, 141-146.

Sheard, J., Simon, Carbone, A, Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Harland, J., Lister, R., Philpott, A., & Warburton, G. (2011). Exploring programming assessment instruments: a classification scheme for examination questions. *7th International Computing Education Research Conference*, 33-38.

Sheard, J., Simon, Dermoudy, J., D'Souza, D., Hu, M., & Parsons, D. (2014). Benchmarking a set of exam questions for introductory programming. *16th Australasian Computing Education Conference*, 113-121.

Shuhidan, S., Hamilton, M., & D'Souza, D. (2009). A taxonomic study of novice programming summative assessment. *11th Australasian Computing Education Conference*, 147-156.

Shuhidan, S., Hamilton, M., & D'Souza, D. (2010). Instructor perspectives of multiple-choice questions in summative assessment for novice programmers. *Computer Science Education*, 20(3), 229-259.

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. *14th Australasian Computing Education Conference*, 61-70.

Terrell, J., Richardson, J., & Hamilton, M. (2011). Using web 2.0 to teach web 2.0: a case study in aligning teaching, learning and assessment with professional practice. *Australasian Journal of Educational Technology*, 27(5), 846-862.

Thomas, R. N., Cordiner, M., & Corney, D. (2010). An adaptable framework for the teaching and assessment of software development across year levels. *12th Australasian Computing Education Conference*, 165-172.

Wang, T., Su, X., Ma, P., Wang, Y., & Wang, K. (2011). Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1), 220-226.

Waycott, J., Sheard, J., Thompson, C., & Clerehan, R. (2013). Making students' work visible on the social web: A blessing or a curse? *Computers & Education*, *68*, 86-95.

Yorke, M. (2011). Assessment and feedback in the first year : the professional and the personal. *14th Pacific Rim First Year in Higher Education Conference*, 1-31.

Understanding the Teaching Context of First Year ICT Education in Australia

Matthew Butler
Monash University
Australia
matthew.butler@monash.edu

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Michael Morgan
Monash University
Australia
michael.morgan@monash.edu

Katrina Falkner
University of Adelaide
Australia
katrina.falkner@adelaide.edu.au

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Amali Weerasinghe
University of Adelaide
Australia
amali.weerasinghe@adelaide.edu.au

Abstract

This paper reports on an investigation of the teaching context of first-year Information and Communications Technology (ICT) courses at Australian universities and the influences of this on students' learning experiences. This is part of a larger project which aimed to identify and disseminate good practices in ICT teaching at Australian universities with a specific focus on the first-year experience. We conducted a systematic review of the research literature from the previous five years and an online search of information on existing courses and content, and interviewed 30 academics concerned with design and delivery of the first-year learning experience in 25 Australian universities. From our study of teaching context we gained a comprehensive view of the current curricula, teaching models and teaching spaces and were able to outline the unique challenges that our first-year ICT students face and to recommend areas for further investigation.

Keywords: First Year; Student Experience; Curriculum; Learning Spaces.

1 Introduction

The transition from secondary to tertiary studies is a difficult process for many students and it is therefore important to understand the influences on this experience. The relatively high rate of attrition in ICT courses indicates that there may be challenges that are unique to this field. While there are a number of studies of the first-year experience across the university sector, to investigate these challenges it is necessary to consider the ICT context. The volume of the literature concerned with specific ICT education indicates that a lot of worthwhile research is being conducted but this research needs to be properly collated and evaluated in order to drive change in practice.

In this paper we report findings of a study that investigated the teaching context in first year Information and Communications Technology (ICT) courses in Australia. The study comprised a review of recent literature on what content is taught in ICT courses, the teaching delivery models used and where the teaching takes place; a survey of Australian university websites; and interviews of Australian academics involved in teaching first year ICT courses. The aims of the study were: 1) to gain an overview of what is taught in first year ICT courses in Australia; 2) to gain understanding of the teaching delivery models used; 3) to gain understanding of where teaching is conducted; and 4) to identify examples of good practice in first year ICT courses in Australia that could be adopted and disseminated widely. This study is part of a larger project of teaching practices in first year ICT courses.

2 Research Approach

This section describes the approach used to investigate the teaching context in the first year of ICT courses. The investigation was part of a project that investigated the broader topic of research and practice in ICT courses in Australia. To conduct the project, the team developed a framework with six themes that together describe the learning experience: "what we teach", "where we teach", "how we teach", "how we assess", "learning support" and "student support". As the focus of this paper is about teaching context, only findings from the "what we teach" and "where we teach" themes will be reported.

The project was conducted in two phases:

Phase 1, Literature review: An examination of current trends and good practice in ICT education nationally and internationally was conducted in the form of a detailed *systematic* review of relevant research literature. The review covered national project reports and key journals and conferences in computing education.

Phase 2, Survey of current practice: Information about ICT courses in Australia was gathered from a survey of university websites. In addition, extensive interviews were conducted with 30 first-year ICT academics from 25 universities in Australia, using an interview script based upon the six themes. All universities that delivered ICT courses were approached. Exemplars of good practice were identified from the the interviews.

2.1 Literature Review

In order to identify current research trends and issues concerning the first-year experience of ICT students in higher education, particularly in the Australian context, a detailed and systematic review of the available literature was conducted. To ensure currency, the scope of the literature was limited to research papers published between 2009 and 2014. Full peer-reviewed research papers published in high-quality academic journals and conferences relevant to the area of study were targeted.

The review began with a series of keyword searches in Google Scholar of relevant terms in the date range from 2009 to 2014. Combinations of keyword searches were carried out in Google Scholar and the searches of combinations of terms continued until no new relevant research papers were being identified. Similar keyword searches were also conducted in the IEEE Xplore and ACM Digital Library databases. In order to ensure that no relevant literature was overlooked, a manual search of selected high-quality research journals and conferences in the area of computing education was conducted for the years 2009-2014.

2.2 Survey of current practice

A survey of current practice was conducted via a survey of online information on ICT courses at all Australian universities and interviews of relevant academics. The purpose of the interviews was to collect detailed information about teaching practices and factors impacting the first-year experience of ICT students in the Australian higher education context. In order to gain this information the project targeted academic staff directly involved in the design, coordination and delivery of first-year courses, as these participants were likely to provide the required insights into the first-year experience and to be in a position to highlight recent changes and examples of good practice.

Participants were selected from each participating university in Australia that delivered an ICT course. Project members nominated relevant people at various universities from their knowledge of the ICT education community. Where this could not be done, the contact details listed on faculty and degree websites were used to initiate e-mail contact. Thirty academics from twenty-five Australian Universities were interviewed. These included six Group of Eight (Go8), three Australian Technology Network (ATN), six Innovative Research (IRU) universities and three Regional University Network (RUN).

The interview script was designed by the project team using the six project themes as a framework. The script consisted of a number of semi-structured questions. The questions related to this paper can be found in the Appendix. The interviewer was encouraged to ask follow-up questions if interesting practices or new issues emerged. The script was trialed in two pilot phone interviews, and slight modifications were made to reduce duplication of the topics covered and to reduce the likely interview time. The revised script was used for all subsequent interviews. Interviewees were sent the list of questions prior to the interview so that they would be aware of the nature of the questions to be covered. All interviews were conducted by telephone during February and March 2014, at a time convenient to the interviewee concerned. A consistent approach was assured by the fact that all interviews were conducted by the same person.

Twenty-nine interviews (one interview involved two participants) were recorded, ranging in duration from 16 to 74 minutes and averaging 53 minutes. Detailed summary notes were taken during each interview. After each interview the notes were elaborated upon and organised into the six themes. The notes were annotated with the approximate times at which the discussion could be found in the audio recording. The interview notes were then examined to find important issues and to identify possible case studies of good practice for further investigation. Detailed quotes from relevant interviews were subsequently transcribed as required. A more detailed description of the methodology used in this project can be found at *Experiences of first year students in ICT courses: good teaching practices*: Final Report: ICT student first year experiences (http://www.acdict.edu.au/ALTA.htm).

The following section reports the results of our investigation into teaching context. We first describe the curricula and curriculum designs of first year ICT courses drawing upon the data gathered from the "what we teach" theme. Following is an investigation of teaching models and teaching spaces drawn from the "where we teach" theme. These themes cover the broad area of the teaching context.

3 What we teach

Our investigation of what we teach focused on the core curriculums of the first year of ICT courses in Australian universities and the process of curriculum design. Relevant courses from all Australian universities were identified and the units offered to first-year students examined to identify similarities between courses and units as well as key areas of differentiation. The teaching of computer programming was explored in detail as this topic is widely researched and discussed in the literature. Also covered in this theme were factors influencing course and unit design, such as the guiding principles adopted from the Australian Computer Society (ACS) and Association for Computing Machinery (ACM)/ Institute of Electrical and Electronics Engineers (IEEE).

3.1 ICT Courses in Australia

A survey of ICT courses in Australian universities found that all but one university (University of Notre Dame) offer an ICT or related degree. While most degree offerings are located in capital cities, a substantial number are offered in rural locations, and a number in off-campus mode.

The faculties that offer ICT degrees are predominantly Information Technology, Science, Engineering, or Business (or faculties that are a combination of these disciplines). There are now very few dedicated ICT faculties in Australian universities. Different ICT degrees are in some cases taught within different faculties in the same university, depending on the context of the degree. For example, a Computer Science degree may be located within an Engineering or Science faculty or department, while an Information Systems degree may be located within a Business faculty or department. In most cases, however, one faculty takes ownership for all ICT-related degrees.

The degrees offered by Australian universities typically fall into one of the following broad categories/contexts:

- general ICT
- ICT with a major or specialisation. Majors typically include
    - games programming
    - software/application development (including mobile)
    - security
    - networks
    - web design and development
    - multimedia
- software engineering
- computer science
- business information systems

General ICT courses, most with majors, make up the majority of courses offered. Computer science ranks second, software engineering third, and information systems / business information systems fourth. There are also a number of miscellaneous ICT courses focusing on other specialist areas such as multimedia, game development, cyber-security and engineering.

In keeping with our focus, we consider units situated in the first year of a typical progression in these courses. Units studied in first year depend on the particular course being taken; however, there is some consistency in units undertaken by students in their first year of ICT study. Common units include:

- programming
- database
- systems analysis
- computing fundamentals
- mathematics (predominantly in computer science courses)

Programming and database are the units most frequently studied by first-year ICT students.

3.2 Literature Perspectives

In the literature search 28 research papers were found related to the theme of 'what we teach' in the context of ICT university courses. Thirteen papers were focused on the first year of ICT courses and ten papers were set in the Australian context. However, only three papers were set in both Australian and first-year contexts (Corney, Teague & Thomas, 2010; Mason, Cooper & de Raadt, 2012; Mason & Cooper, 2014) and all three of these papers relate specifically to programming.

Approximately half the papers found discuss high-level curriculum design issues. These papers typically present guides and frameworks for using noted ICT charters (such as ACS, ACM, IEEE, and SFIA) in curriculum design, often highlighting specific case studies of recently redesigned curriculums (Adegbehingbe & Obono 2012; Koohang et al, 2010; Herbert et al, 2013a). Because of this, the literature on curriculum is often not focused on the first-year context. While discussion of curriculum design can identify certain needs for structuring courses with supporting progressions, these papers typically discuss design of an entire three- or four-year curriculum.

Moves to adopt SFIA in curriculum design are evident in the more recent papers. Several Australian universities appear to have adopted this framework as a key charter in redesigning their curriculums, with the University of Tasmania being a well-documented example of this (Herbert et al, 2013a; 2013b; 2013c; 2014). The SFIA framework is of importance in its presentation not only of core skills as they relate to industry but also of levels of responsibility, which can be aligned to different year levels in a course (von Konsky, Jones & Miller, 2014). Consequently, these papers provide some insight into curriculum design within the first-year context.

The publications relating most closely to the first-year context deal with narrower fields of study within the first year. For example, discussion of programming curriculum and issues in most cases relates specifically to novice programmers, thus usually the first-year context. Indeed, programming was clearly the most represented context, with 11 papers relating specifically to curriculum issues within this area of study. Mason, Cooper & de Raadt (2012) and Mason & Cooper (2014) provide a comprehensive analysis of trends in introductory programming courses in Australian universities. They note a fragmentation of choice of the programming language being used, and a reduction in the use of Java as a language in introductory programming courses. Issues raised by other researchers relate mainly to the choice of programming language and environment (Fincher et al, 2010; Stefik & Siebert, 2013), and restructure of curriculum to better support novice programmers (Corney, Teague & Thomas, 2010; Hu, Winikoff & Cranefield, 2013; Thota & Whitfield, 2010). The narrower focus suggests that notions of what we teach are more easily placed in the context of a specific year and unit, while broader curriculum issues (both design and content) will focus on whole courses.

Other specific contexts for discussion of curriculum issues were found, although much less prevalent than those relating to programming. Subject areas found include computer systems (Benkrid & Clayton, 2012; Patitsas et al, 2010) and software development (Thomas, Cordiner & Corney, 2010). Other sub-themes that were found in the literature relating to curriculum include investigation of gender issues (Koppi, Roberts & Naghdy, 2012) and career progression and its implications for curriculum design (von Konsky, Jones & Miller, 2014).

In summary, there is little recent literature about what is taught to first-year students in the Australian context. While there is research relating to curriculum development in higher-education ICT courses, it tends not to address specific first-year issues, which are typically reported on in relation to specific topics such as programming. This suggests that there is scope for further research relating to how curriculum is developed in consideration of the needs of first-year students.

3.3 Current Practice in Australia

The interview questions related to the theme of 'what we teach' sought added insights into the nature of first-year ICT courses in terms of student demographics, the development of the teaching curriculum and, more specifically, programming languages taught.

Demographics of first year of ICT courses

Enrolments in the first year of ICT courses vary considerably across Australia, ranging from approximately 100 to 500 students. According to interviewees it is often difficult to gauge exactly how many students are in the first year of a course, as different students enter the courses by different pathways, some of which will attract credit for designated units. Many interviewees made informed estimates of the numbers on the basis of enrolment numbers in units that were core for first-year students, along with the course information of those students. Based on the interviewees' responses, just over 5000 first-year students were estimated to be enrolled in ICT courses across the 25 universities contacted.

The mix of students also varied considerably across the universities. Many interviewees were not privy to the breakdown of local versus international students, but most were able to give informed estimates, again based on class demographics. In view of the uncertainty of these estimates, we present only the broad picture. Six institutions indicated very low numbers of overseas students (less than 10%), while another six indicated that 50% or more of their first-year cohort were international students. Between these extremes, the majority of interviewees (7) estimated their international enrolments as 20-30% of their cohorts. There would appear to be scope for research into the internationalisation of the teaching curriculum, not only because of these demographic estimates, but also because of the international nature of ICT.

Curriculum design

Interviewees were asked whether the design of their courses was influenced by any external curriculums. Most interviewees indicated that their courses are accredited by the Australian Computer Society. Many mentioned that their course designs were influenced or inspired by external bodies such as the ACS, ACM, and IEEE as well as industry companies like CISCO. Although these organisations played an important role in the consideration of their curriculum design, interviewees were often unsure exactly how the frameworks provided by these organisations were specifically used. An illustrative response:

"The degree programs are a combination. It is not directly taken from the ACM/IEEE computer science curriculum but they were used as input into the design of the course. So we used the ACM/IEEE curriculum as well as the ACS guidelines. The courses are ACS accredited." (U1)

There is little literature on the exact role of bodies such as ACS, ACM, and IEEE in curriculum design, suggesting an opportunity for research to seek greater insights into the role of such formal bodies in the design and development of the tertiary curriculum.

The use of SFIA in curriculum design was notably absent from the interviews. Recent literature suggests that it can play a major role in the design of courses, so it was of interest that it was not mentioned by any interviewees. This is likely to change in the near future, as SFIA gains awareness through both the ACS and published literature.

Programming languages

Interviewees were asked what programming languages are introduced to students in their first-year ICT courses. The most common languages were Java (16) and Python (12). Java has been well documented as a language used to teach students programming both at a foundation level and also as an introduction to object-oriented programming. While Java remains a popular choice, a number of interviewees reported recent moves away from Java as an introductory language, in many cases to Python. Interviewee U4 explained this shift in languages:

"Java was seen as having too much excess baggage to get people off the ground that just wanted to learn the basics. They didn't go into object-oriented or object-based programming so the need for all of the concepts around object-oriented programming weren't necessary and so instead they wanted to build the strength in the fundamentals and the wisdom was that Python would be better."

Another interviewee echoed these sentiments, noting that:

"We are considering at the moment moving away from Java and maybe going to something like Python. We've used Java for a fair while but it's losing relevance in a lot of areas and is a quite bloated language. Something like Python is more elegant and sophisticated in some ways and enforces some good program structure and at least as good at formatting, so it's better for the first-year students to introduce them to the programming concepts." (U6)

In contrast, interviewee U7b indicated a move from C++ to Java as the introductory programming language, *"Changed from C++ to Java, very popular in industry, slightly easier."*

Concerns have been raised in the literature about the significant learning challenges faced by novice programmers starting with an object-oriented language such as Java, and some responses in the interviews appear to address these concerns. While a number of interviewees discussed their shift to Python, others had moved to less traditional languages and environments such as Processing, Gamemaker, and Scribble (a variant of the Scratch programming environment). The literature also includes the move to environments such as Alice. These examples appear to place the emphasis on problem solving rather than language syntax or complex programming paradigms; however, little research has been found that describes the learning outcomes of these changes.

One interviewee said that the move from Java to Scribble, a visual programming language, was to "*get students to focus on solving problems rather than concentrating on syntax*" (U15b). A program is constructed in Scribble by assembling visual blocks representing code segments, a process that shields novice programming students from syntax and code and allows them to focus on programming logic. This is seen as a more accessible environment than a traditional programming language for introducing fundamental programming concepts to novice programmers. As interviewee U15b explains:

"It was a fair undertaking, and it was a fairly big decision to say let's not start students in a syntactic language like Java. I mean there is always the question of which language do you choose. So it was a very concerted effort to get away from that and to say no we need to focus on creating problem solvers first."

Interviewee U15b observed that the student evaluations for the unit have been really good, but the important consideration is how the students will perform in subsequent units. Students study at least one more programming language in their course, for example, Python, Java or C++. The transition to these subsequent programming units is currently of some concern, and the effects of the change are currently being formally evaluated.

The introduction of programming languages focused on mobile development platforms is a relatively recent inclusion in the programming curriculum prompted by current industry trends. Interviewees U24 (two interviewees were involved in this interview at the same time) described the introduction of Objective C and XML as the programming languages for smartphone/tablet development in iOS:

"We actually have started introducing some new programming languages. We now include Objective C .... We now also teach XML and we've introduced smartphones and iPads into our learning space too."

This further demonstrates the diversity of approaches that are currently being explored in introductory programming units. *"We introduced the Mac to replace the tablet PCs two years ago and they were introduced so we could teach iOS languages."* In part this change was made to appeal to students by targeting a computing environment, in the form of mobile devices such as smartphones and tablets, with which the students engaged on a regular basis. In terms of research, a formal evaluation and comparison of the range of approaches currently being trialled in the Australian context would be of benefit.

Some universities place the introduction to programming into a web development context, using web-scripting languages such as Javascript and HTML. Other languages mentioned included Visual Basic, C, C# and ActionScript (Flash). One interviewee indicated that a number of languages are covered across their degrees, but not in the first programming unit:

"What we do in the first semester. We teach it in a language neutral fashion... We deliver the material in language neutral fashion so it's about the programming concepts not specifically about the one language. We teach them the way to do something in general not in a particular language. Then we have material that helps them learn how to apply those concepts in a particular language." (U1)

3.4 Summary

What the literature and especially the interviews highlight is that there appears to be little consensus as to what programming language or environment best supports novice programmers. Many institutions recognise the inherent difficulties for novice programmers, but the quest for the ideal learning approach appears far from over.

The study of curricula and curriculum design provides a background for our investigation of teaching context in terms of teaching models and teaching spaces.

4 Teaching Context

Our investigation of teaching context was drawn from the 'where we teach' theme which focused on the teaching models and teaching and learning spaces used for first-year ICT courses in Australian universities. It considered the design and use of new teaching spaces and the redesign of existing spaces, either physical or virtual. For virtual teaching spaces, the theme included teaching and learning in situations enabled through the use of mobile and ubiquitous technologies.

4.1 Literature Perspectives

The systematic literature review found 13 papers that were concerned with the 'where we teach' theme. All of the papers were set in the higher education sector and in the context of programming – all but one of them in introductory programming; two were Australian studies. The papers found for this theme report studies of a variety of different teaching and learning spaces. Govender (2009) explored the lecture setting in an investigation of the influence of the learning context on how students approach the task of learning to program and their ultimate success. Cheryan, Meltzoff & Kim (2011) investigated the effect of virtual learning environment design on male and female students' interest and anticipated success in an introductory computer science course. Both studies concluded that context was an important factor in students' success in learning to program.

A study by Howles (2009) compared the impact of different learning environments on student retention. The findings revealed that a change from a studio environment (20 students with access to computers) to an active learning environment (40 students without computers) did not negatively impact student retention.

Australian researchers Alammary, Carbone & Sheard (2012) describe the implementation of a virtual 'smart lab' for assisting programming lab class teachers. The smart lab monitors students' progress as they perform programming tasks, enabling instructors to readily respond to individual students and assess the overall progress of the class. An evaluation demonstrated the usefulness of the smart lab in providing timely and appropriate feedback to the teachers. Another Australian study by Maleko, Hamilton & D'Souza (2012) explored novices' perceptions and experiences of a mobile social learning environment designed to enhance student-to-student interactions. A key finding of this study is that most students engaged more with their learning and with colleagues in the mobile social environment than in the face-to-face environment. Small learning communities were formed, enabling students to interact regardless of their physical location or the time of day.

Considerable resources have been expended on the development of environments to support the teaching and learning of programming, and a number of these have been specifically designed for introductory programming students. There are many studies of the use of these environments for engaging students in the learning process and helping them to learn to program. Verginis et al (2011) studied a web-based learning environment, SCALE (Supporting Collaboration and Adaption in a Learning Environment), and found it valuable for supporting learning in introductory computer science. Moons and De Backer (2012) present an interactive programming environment, EVizor (Educational Visualization of the Object Oriented Run-time), implemented as a Netbeans plugin. The EVizor system visualises program execution and incorporates explanations and embedded quizzes. The system design is founded on constructivist and cognitivist learning theories. A series of evaluations and experiments showed that it is useful in helping students understand program behaviour.

Fincher and Utting (2010) introduce Alice (Cooper, 2010), Scratch (Maloney et al, 2010) and Greenfoot (Kölling, 2010), three environments widely used in introductory programming courses, each of which has a different focus and approach. The design rationale and pedagogical approach that each supports are explained in a series of articles by the designers. Wellman, Davis & Anderson (2009) introduced Alice into an introductory programming course to increase students' interest in computer science. They report that students were motivated and engaged in the learning activities. However, Garlick and Cankaya (2010) had a different experience. In an experimental study they found that students who used Alice in their introductory programming course had lower performance and responded less favourably compared to students who were given traditional instruction.

In summary, there are very few examples of recent literature discussing the first-year ICT learning environment in the Australian context, therefore further research is needed in this area. Current research focuses on specific examples of virtual lab software, the inclusion of social networking tools to promote learning communities, web-based collaborative learning environments, and a variety of introductory programming environments. There is a need to conduct further research on both physical and virtual learning environments that are tailored to the needs of first-year students in the ICT context.

4.2 Current Teaching Context in Australia

The interview questions related to the teaching context sought detailed information about teaching spaces in Australian universities and how they are used. In addition to describing the physical teaching spaces, interviewees were asked to provide information about their teaching in online or blended environments. Their responses gave insights into current teaching models and into the physical and virtual spaces where teaching is conducted. The responses to these questions are discussed under the main topics that were identified from the analysis of the interview data.

Teaching models

An important factor in a discussion of 'where we teach' is the teaching model that is used. The most common teaching models used in the universities in our study are the traditional *lecture/laboratory* and *lecture/tutorial/laboratory* combinations. However, there were indications that a number of institutions had moved or were in the process of moving to different models, often involving a shift from physical to virtual teaching spaces. Many interviewees mentioned recent changes to lectures. Interviewee U21 described a radical change where a new degree has been implemented with only a single introductory lecture. Subsequently, students are provided with audio video clips and a text book in paper or electronic form. Tutorial classes are either on-campus or online.

A number of interviewees indicated that the teaching time devoted to lectures has been reduced. For example, interviewee U10 stated:

"So we used to have a very standard model of 3 lectures a week and 1 practical session and then we moved it to 3 lectures a fortnight and 1 practical session and 1 collaborative workshop session every week."

In another example interviewee U7b indicated that they had:

"Cut down lecture 2 hours to 1, less talking at the students, the boring stuff. Gone with a tutorial and a practical session, more hands on stuff particularly for the first-years."

In addition, "*All recordings lectures and materials go onto an online Blackboard forum*," so students can access them when convenient.

Several interviewees mentioned the reduction of lecture time in order to increase practical lab sessions. For example, interviewee U24 commented:

"first-year programming a special case. ... Combined lecture and practical into a workshop. For online students they submit weekly tasks to the lecturer and she checks and gives feedback within 24 or 48 hours".

In this case the lecturer combined the traditional lecture and practical session into a 3- or 4-hour session (2 hours, a 1-hour break, then another 1 or 2 hours) and called it a workshop. Interviewee U24 observes enigmatically that "*Workshop mode equals flipped classroom minus the pre-class activities*." Although the reduction in lecture time and the corresponding increase in practical sessions was seen to be more resource-intensive it was also seen to be more productive in terms of increased student engagement and therefore increased student retention.

The most common teaching innovation discussed by interviewees was *blended learning*, and this was having an influence on the way teaching space is used. From the interviewees' comments, however, it is apparent that there are various understandings of the term 'blended learning' and a variety of ways in which this teaching model is implemented. A couple of interviewees used the term to mean the provision of online resources to both on-campus and online students. Several interviewees were exploring the 'flipped classroom' model, where the homework and class activities are reversed. Interviewee U18 said that first-year students had reacted negatively to this teaching model. She felt that the first-year students were not organised enough to watch the videos on their own and she questioned the suitability of this model for first-year students. In a more extreme example, interviewee U7a indicated that they favoured "*Small lectures, big tutorials. Light presentation and heavy practicals*." They indicated that they had *"Removed face to face lectures, some years ago"* and placed "*More emphasis on tutorials with the support of online modules using videos"*. U7a further explained that "*Students need to look at video lectures and background readings before [the] tutorial*."

Physical teaching spaces

Interviewees gave descriptions of their various physical teaching spaces. Lectures are typically held in theatres with capacities ranging from 100 to 400 students. Tutorials are usually held in classrooms holding 30 to 40 students. Laboratory classes are typically held in computer labs with space for 20 to 30 students, although a couple of interviewees mentioned labs of 40 to 50 students.

Most interviewees agreed that lecture theatres are less than ideal teaching and learning spaces. Many interviewees raised the issue of lack of student attendance at lectures. While there is a general shift towards reducing time spent in lectures or replacing lectures with more practical classes, there is also a considerable effort being made to improve the learning experience in lectures. Some have introduced new teaching models for lectures and others employ a variety of techniques to motivate and engage the students.

Recording of lectures is now commonplace, with half the interviewees indicating that all lectures are recorded at their institution. Some interviewees stated that lecture recording is mandatory while others mentioned an opt-out policy. At a couple of institutions, where lecture recording systems are not readily available, some individuals record their own lectures. Only a couple of interviewees do not record their lectures in some way. The most common recording system is Echo360; others in use are Blackboard Collaborate and Lectopia. The availability of lecture recordings (and in some cases tutorial classes) has reduced the impetus for students to attend on-campus.

Most innovation in the design of physical teaching spaces is apparent in the computer labs where practical classes are held. Computer labs are traditionally set up with straight rows of tables and a computer for each student. At a couple of institutions there are variations on this arrangement. In one institution the lab has multiple fronts and in another the computers are placed around the four walls of the lab with the teacher in the centre. However, a number of institutions have made more radical changes to their computer labs, redesigning them into collaborative learning spaces. One interviewee described a room with tables seating 4 to 6 students, each with a large screen and one keyboard. Another described a similar teaching space with facilities for displaying the work of each group on a central screen for the whole class to view. Some of these labs hold more students than traditional labs and have been designed as flexible learning spaces.

A few interviewees mentioned more radical designs in teaching spaces. At one institution there are dual teaching spaces where students can move from a classroom setup to a computer lab in a large room divided by a partition. Another, smaller, institution uses only one type of teaching space. The room seats 50-60 students at eight sets of reconfigurable tables. This flexible teaching space has multiple fronts with a data display unit, fixed and mobile white boards and multiple power points around the perimeter of the room and hanging from the ceiling. One interviewee, describing a radical shift away from the traditional teaching model to a blended learning model, said that their learning spaces include "*libraries, site inspection and even corridor meeting, tearooms and virtual teaching environments*" (U7a).

Virtual teaching spaces

Some interviewees acknowledged the increasing importance of virtual teaching spaces. Online learning is happening at most institutions, either with units taught only in online mode or with units taught online in combination with on-campus teaching. A number of interviewees mentioned small cohorts of online students in their on-campus units. Several indicated that all their units are available both on-campus and online, with students having access to teaching resources made available to both cohorts. They saw no difference between the resources provided to their on-campus and off-campus students. As interviewee U24 commented:

"I think we have two main teaching spaces – one is the physical space and one is virtual space. The virtual space is constructed with as much care to the design as the physical space is."

All institutions use a form of Learning Management System (LMS) where typically all course materials are placed. The most commonly used LMS are Blackboard and Moodle. A couple of interviewees emphasised that these are not really learning environments but just delivery platforms for course content. One institution uses Captivate Workshop for delivery of learning objects. A couple of interviewees mentioned other online environments developed for use in specific courses. ViLLE (a visual learning tool) is a collaborative education platform developed specifically for learning programming, and IVLE (Informatics Virtual Learning Environment) is an online interactive instructional system for use in teaching programming and algorithmic problem solving.

4.3 Discussion

The aims of the study were: 1) to gain an overview of what is taught in first year ICT courses in Australia; 2) to gain understanding of the teaching delivery models used; 3) to gain understanding of where teaching is conducted; and 4) to identify examples of good practice in first year ICT courses in Australia that could be adopted and disseminated widely. A key finding from our investigation of what is taught in first years ICT courses was that there is little consistency with regard to the programming languages that are introduced to new programmers in ICT courses. While Java and Python are very prominent across the universities of the Australian academics we interviewed, there appears to be no consensus on the best approach to take with novice programmers. This is also reflected in the literature, with research often highlighting the problematic nature of introducing both programming concepts and syntax.

There has been a perceptible trend towards programming environments where the focus has moved away from syntax to problem solving. This is an area that needs investigation to determine how students respond to learning programming in these environments.

Further scope for research is in the use of formal skills frameworks provided by organisations such as ACS, ACM and IEEE. There is little literature and little understanding by the interviewees of exactly how course curriculums are developed with these frameworks in mind. There are a number of recent publications regarding SFIA and its role in curriculum development, and literature such as this may present an opportunity for more formal acknowledgement of these frameworks in this area.

Our investigation of the literature on teaching context found little specific research on the physical and virtual learning spaces tailored specifically for the needs of first-year ICT students in the Australian context. This contrasts strongly with the significant changes to practice highlighted by the interviewees, including changes to the balance between lectures and practical labs and the changing nature of the layout of computing laboratories. A prominent topic raised by interviewees was the design and use of teaching spaces to engage students in active learning experiences. The layout of physical teaching spaces was reported to be increasingly diverse and flexible. Various new physical and virtual learning environments are tailored to the needs of first-year ICT students. Further research is needed to assess the impact of these changes to the teaching environment on student performance and on the student experience.

There were strong indications from the interviewees that the provision of online resources is more prevalent, resulting in an increase in flexible study options, including the integration of social networking tools to assist the formation of student learning communities. These changes highlighted the need for further research in order to assess their impact on the first-year ICT student experience.

5 Conclusion

Our investigation of teaching context in first-year ICT courses in Australia has highlighted many new initiatives in teaching delivery models and the design of teaching spaces, driven largely by a desire to provide interesting learning environments and active learning experiences. The research has identified the need to undertake further research investigating such areas as curriculum design, development of graduate attributes, and understanding the needs of the ICT industry. An imperative now is also to assess the effectiveness of the innovations identified in engaging students and enhancing their learning. Evidence from such evaluations is essential for promotion of these innovations and driving change in the ICT teaching sector.

6 Acknowledgements

This project was undertaken with the support of the Australian Council of Deans of Information and Communication Technology through the ALTA Good Practice Reports Commissioned for 2013–2014 grant scheme (http://www.acdict.edu.au/ALTA.htm).

The project team would like to acknowledge the work of Beth Cook who worked as a research assistant to conduct the interviews and to prepare the detailed interview notes.

7 References

Adegbehingbe, O. D., & Eyono Obono, S. D. E. (2012). A framework for designing information technology programmes using ACM/IEEE curriculum guidelines. *World Congress on Engineering and Computer Science 2012*.

Alammary, A., Carbone, A., & Sheard, J. (2012). Implementation of a smart lab for teachers of novice programmers. *14th Australasian Computing Education Conference,* 121-130.

Benkrid, K., & Clayton, T. (2012). Digital hardware design teaching: an alternative approach. *ACM Transactions on Computing Education*, *12*(4), 13.

Cheryan, S., Meltzoff, A. N., & Kim, S. (2011). Classrooms matter: the design of virtual classrooms influences gender disparities in computer science classes. *Computers & Education*, 57(2), 1825-1835.

Cooper, S. (2010). The design of Alice. *ACM Transactions on Computing Education*, *10*(4), 15.

Corney, M., Teague, D., & Thomas, R. N. (2010). Engaging students in programming. *12th Australasian Computing Education Conference*, 63-72.

Fincher, S., Cooper, S., Kölling, M., & Maloney, J. (2010). Comparing Alice, Greenfoot & Scratch. *41st ACM Technical Symposium on Computer Science Education*, 192-193.

Fincher, S., & Utting, I. (2010). Machines for thinking. *ACM Transactions on Computing Education*, 10(4), 13.

Garlick, R., & Cankaya, E. (2010). Using Alice in CS1: A quantitative experiment. *15th Conference on Innovation and Technology in Computer Science Education*, 165-168.

Govender, I. (2009). The learning context: Influence on learning to program. *Computers & Education*, 53(4), 1218-1230.

Herbert, N., Dermoudy, J., Ellis, L., Cameron-Jones, M., Chinthammit, W., Lewis, I., de Salas, K. L. & Springer, M. (2013a). Stakeholder-led curriculum redesign. *15th Australasian Computing Education Conference*, 51-59.

Herbert, N., Lewis, I., & Salas, K. De. (2013b). Career outcomes and SFIA as tools to design ICT curriculum. *24th Australasian Conference on Information Systems*, 1-10.

Herbert, N., Salas, K. De, Lewis, I., Cameron-Jones, M., Chinthammit, W., Dermoudy, J., Ellis, L. & Springer, M. (2013c). Identifying career outcomes as the first step in ICT curricula development. *15th Australasian Computing Education Conference*, 31-40.

Herbert, N., Salas, K. De, Lewis, I., Dermoudy, J., & Ellis, L. (2014). ICT curriculum and course structure : the great balancing act. *16th Australasian Computing Education Conference*, 21-30.

Howles, T. (2009). A study of attrition and the use of student learning communities in the computer science introductory programming sequence. *Computer Science Education*, 19(1), 1-13.

Hu, M., Winikoff, M., & Cranefield, S. (2013). A process for novice programming using goals and plans. *15th Conference on Innovation and Technology in Computer Science Education*, 3-12.

Koohang, A., Riley, L., Smith, T., & Floyd, K. (2010). Design of an information technology undergraduate program to produce IT versatilists. *Journal of Information Technology Education*, 9, 99-113.

Koppi, T., Roberts, M., & Naghdy, G. (2012). Perceptions of a gender-inclusive curriculum amongst Australian information and communications technology academics. *14th Australasian Computing Education Conference*, 7-14.

Kölling, M. K. (2010). The Greenfoot programming environment. ACM *Transactions on Computing Education*, 10(4), 14.

Maleko, M., Hamilton, M., & D'Souza, D. (2012). Novices' perceptions and experiences of a mobile social learning environment for learning of programming. *17th Conference on Innovation and Technology in Computer Science Education*, 285-290.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4), 16.

Mason, R., & Cooper, G. (2014). Introductory programming courses in Australia and New Zealand in 2013 – trends and reasons. *16th Australasian Computing Education Conference*, 139-147.

Mason, R., Cooper, G., & de Raadt, M. (2012). Trends in introductory programming courses in Australian universities: languages, environments and pedagogy. *14th Australasian Computing Education Conference*, 33-42.

Moons, J., & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, 60(1), 368-384.

Patitsas, E., Voll, K., Crowley, M., & Wolfman, S. (2010). Circuits and logic in the lab: toward a coherent picture of computation. *15th Western Canadian Conference on Computing Education*, 7.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4), 19.

Thomas, R. N., Cordiner, M., & Corney, D. (2010). An adaptable framework for the teaching and assessment of software development across year levels. *12th Australasian Computing Education Conference*, 165-172.

Thota, N., & Whitfield, R. (2010). Holistic approach to learning and teaching introductory object-oriented programming. *Computer Science Education*, 20(2), 103-127.

Verginis, I., Gogoulou, A., Gouli, E., Boubouka, M., & Grigoriadou, M. (2011). Enhancing learning in introductory computer science courses through SCALE: an empirical study. *IEEE Transactions on Education*, 54(1), 1-13.

von Konsky, B. R., Jones, A., & Miller, C. (2014). Visualising career progression for ICT professionals and the implications for ICT curriculum design in higher education. *16th Australasian Computing Education Conference*, 13-20.

Wellman, B. L., Davis, J., & Anderson, M. (2009). Alice and robotics in introductory CS courses. *5th Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations*, 98-102.

8 Appendix

Below are the indicative interview questions used to capture current practice regarding student demographics, curriculum, and teaching spaces:

Demographics

- What undergraduate computing degree(s) do you offer?
- In which faculty? Or are they multi-faculty?
- How big is the first-year cohort? (We agreed that we were talking principally here about Australian campuses, though some respondents with overseas offerings might also mention those.)
- What's the demographic profile of the students (overseas / domestic / distance / full-time / part time)?

What we teach

- What ICT courses/subjects/units are offered to first-year students? Briefly describe the content of each course.
- What programming languages are taught? What other software packages are taught?
- Is the content of these courses based on some external curriculum, such as the ACM/IEEE curriculum, or more on your group's own design?

Where we teach

- Describe your teaching spaces.
- In addition to physical teaching spaces, what teaching do you do in blended or online environments?
- Have you made any changes recently (in the past 5 years)? What? Why? Has it worked?
- How do you know (evaluation)?

Considerations in Automated Marking

Joel Fenwick

The University of Queensland,
Centre for Geoscience Computing
QLD 4072
Australia
joelfenwick@uq.edu.au

Abstract

With large classes and high demands on the time of teaching academics, (as well as the need to keep marking budgets under control) evaluating the functional correctness of programming assignments can be challenging. Entirely automating the evaluation process may seem desirable but that would deny students formative feedback from more experienced programmers. This in turn reduces their opportunity to correct errors in their practice.

Instead, this paper contains a discussion of marking processes where much of the "heavy lifting" or repetitive work is automated but still allows for human feedback. We discuss the impact of automated marking on assessment design, students, and where the hard work is hidden.

The literature contains descriptions of many projects for automating various parts of the process with varying interfaces and levels of integration with external systems. In the author's opinion though, that they are not strictly *required*, and we describe a simpler set of requirements.

Keywords: Programming Assessment, Automated Marking, Assessment Design

1 Introduction

Programming assessment submissions can be evaluated in a number of ways. They can be judged on how well they have implemented specified functionality (either by direct testing or by inspection); how readable and well structured their source code is; their algorithmic complexity or runtime performance; or their design and the process used to produce them (typically in more advanced courses). In introductory and intermediate courses, the focus tends to be on the first two. Marking large numbers of such assignments requires significant amounts of time to do well and risks uneven treatment as markers tire.

While there is still a need for human judgement when it comes to evaluating things like readability, repetitive testing of functionality seems to be a suitable target for automation. Section 2 looks briefly at the history and development of automation of programming marking. However, whether existing automation packages are adopted or ad-hoc tools are employed, not every task which humans can mark is suitable for simple automation. In this discussion, we focus on black box testing of programs to be submitted and evaluated against some "well defined" specification (as opposed to more open ended "do something cool" type assignments). We will also assume that the functionality of the program (rather than the precise algorithm used to produce it) is the main point of interest.

Section 3 gives more detail about the course which provides the main context for this work. Section 4 gives core questions to be asked when evaluating the functionality of a programming assignment submission. Section 5 gives an example of simple automation work flow in terms of basic primitives. While arbitrarily complex ad-hoc solutions are possible, we limit the discussion to what can be achieved with simple tools and small amounts of custom coding.

Sections 6 and 7 discuss the impacts of this style of automation on students and on the design and description of programming assessment. Section 8 looks at *some* security/integrity considerations. Section 9 concludes with a summary of where the work hides when this type of automation is employed.

2 Some history

Very early work was done by Hollingsworth (1960), who described automated marking for a class of 80 students. A few years later, Forsythe & Wirth (1965), followed suit. There are quite a few common features between these works: The technical details of getting the tests to run are a significant issue. In order for these early graders to work, the student programs must have a particular structure (a trait mirrored in later "unit test" style testers). The authors of both systems acknowledge the possibility of student code interfering with the testing infrastructure but do not consider it to be a major issue. Both systems require manual intervention in the case of badly behaved programs. Douce et al. (2005) would later classify these types of systems as first generation systems.

Deimel & Clarkson (1978) discuss the merits of running student assignment submissions against unseen test data. Gathering student submissions was still an issue at this point. They also state a challenge which is still with us today: that students' real goal 'irrespective of the problem statement, is to produce "correct" output for the supplied input'.

Later, Benson (1985) described a system where students would use email to submit files. These would then be processed against a batch of tests with the results being available to students the next day. This was done before the deadline so that students had a chance to fix errors. These tests were made available "several days before the due date." More detailed tests were used to determine marks once the deadline passed. An interesting approach adopted by Benson was that students could appeal their marks if they could demonstrate that proper testing on their part could not have detected the fault.

Harris et al. (2004) took the pre-testing approach even further with a system where assignments could not be submitted at all unless they passed a set of supplied tests. This requirement was enforced by the submission tool itself. This way only fully functional assignments are considered for further grading.

In this document, we will refer to automated tests accessible before the assignment deadline as *public* tests. Whereas tests used for marking will be denoted *hidden* tests. In the interests of transparency, this second set of tests should also be revealed eventually. As an aside, it is also possible not to provide sample tests, but instead to make the production of tests by students, part of the assessment (Edwards 2003).

The survey by Douce et al. (2005) divides automated testing systems into three generations. First generation systems which relied on technical tricks to operate. Second generation systems used tools already available from the operating system. Third generation systems made use of the web and included a wider variety of testing approaches.

One example (from many) of these third generation type systems is BOSS (Joy et al. 2005). It provides both a web interface and network client application to allow students to lodge code to be tested. This allowed both student testing against a public test set with the possibility of re-submission and testing against the hidden set for staff assessing final submissions. Of particular note here, is the remoteness of the testing. The tool is accessed from a client machine (eg student laptop) but the tests are executed on a server.

Ihantola et al. (2010) carried out a follow up survey covering the years 2006–2010. They identify classifications of testing: using a framework such as JUnit; comparing the output of running programs; scripting the build, test and comparison; and experimental approaches. They concluded "too many new systems are developed" but that a reason this occurred was that tools were complete enough to meet the needs of the course they were created for but not necessarily general enough to be applied elsewhere[1].

2.1 Isn't this a solved problem?

After literature spread over 50 years, isn't automation of programming assignment marking a solved problem by now? Not really, no. The developments behind the generations described in the 2005 survey are not monotonic improvements towards a fixed goal. Moving from the first generation to the second, the ability to construct and run tests at all became less of a problem because there was now greater support from the operating system. Writing comparison based tests became simpler. Similarly, there are less troublesome ways to gather assignment submissions than collecting punch cards or individual emails. Some systems incorporate the submission mechanism (eg BOSS).

The main characteristics of interest in the third generation are greater levels of integration with other systems, and other interfaces to the testing system (typically web interfaces). Systems in this generation aim for reusability between assessments and applicability to a variety of languages. There is still work to be done here though. Tests must still be designed for each assessment (even if just in the form of *input:output* pairs). Also language flexibility typically means one of three options:

- The system already has some support for the chosen language.
- Output matching is being done at a level where the language is irrelevant. For example: capturing text output at the OS level; examining file contents after a run is completed or exchanging messages across a network.
- The system provides an interface to write plugins or subclasses for the chosen language (eg GAME (Blumenstein et al. 2008)).

In their survey, Ihantola et al. (2010) distinguished between automation for marking programming competitions versus "systems for (introductory) programming education." While competition marking is an interesting area, formative feedback does not seem to be a consideration there. On the education side, the parenthesis around "introductory" are important here. More advanced courses have additional requirements or make use of lower level features which are not needed in introductory courses. For example, Solomon et al. (2006) describe the LinuxGym tool for assessing and training students in the use of shell and scripting. They draw a distinction between LinuxGym and BOSS (Joy et al. 2005) due to the fact that their tasks require modification of system state rather than producing output.

2.2 What about xUnit?

A number of automated testing systems (eg BOSS) can make use of libraries from the "xUnit" family. These include PyUnit for Python and JUnit for Java and are derived from a Smalltalk testing library written by Kent Beck (Fowler 2014, Python developers 2014*b*, JUnit project 2014). Individual tests are written as methods of classes which inherit from a class in the xUnit library. These methods can throw exceptions to indicate that a test has failed. After a batch of tests has run, a report can be presented indicating which tests passed and which failed.

In the author's experience, with well written tests, xUnit is an effective means to test an API. In assignments however, this would indicate a library is being written or where the internals of the code have been specified. For example, the assignment is written as: you must write a class X which has

- a method `int thing(int x, int y, int z)` which returns the median of its arguments.
- a method `String meth(String a, String b)` which ...

In more advanced assessments, it may not be desirable to specify implementation details at this granularity. Students could be expected to make their own design decisions rather than be constrained by tests of internals. In these cases, tests using reflection might not naturally fit with specified functionality. They also do not test the external interface. An additional test interface would need to be specified. Now, it is possible to use the xUnit structure to describe tests against external programs (the test functions can contain arbitrary statements in the relevant language), but there does not seem to be any special advantage in doing so.

[1]They also note that experimental systems tend to disappear from the web.

3 Context

The driver for this work is a course with the dual purposes of teaching systems programming concepts and improving programming skill. The previous run of the course had over 300 students and the current offering has over 400. The assessment consists primarily of traditional programming assignments. Marking has two components: functionality ($\sim 85\%$) and style ($\sim 15\%$). The functionality mark is based solely on whether the program produces the correct results and system interactions. It must not only say the right things but also not leave processes running or consume unacceptable amounts of system resources. But, apart from a criterion of not "taking too long" to run, the performance of the algorithms used is not a concern. This part of the marking is done entirely with automated black box testing using simple `bash` scripts. While a single staff member needs to check on the process occasionally, performing this part of the marking is fairly undemanding.

The style component requires attention from human markers who grade submissions on clarity, structure and adherence to a supplied style guide. However, the relatively small fraction of the overall marks means that fine gradations in readability and structure are not required and markers don't need to spend a lot of time doing it. This process ensures that students still receive feedback about how humans read their code.

While the course doesn't go as far as Harris et al. (2004) in rejecting submissions which aren't functionally perfect; submissions which don't pass at least some functionality tests are not marked.

The assignment tasks typically consist of sequences of interactions or commands for users. For example, various card or grid based games; agents interacting with a simulation environment; or system automation. This requires student programs to both be able to recognise valid interactions and to maintain state.

As discussed later, this lends itself to rubrics where marks for more complex tasks depend on successful completion of earlier subtasks. For example: "make a single valid move" leads to "play a complete game".

Assignment submission is done by committing code to a version control repository (`subversion` in this case). This neatly handles re-submission and time stamping as well as exposing students to professionally useful tools. Gathering submissions for marking only requires: a list of students, two version control commands[2] and a `bash` *for* loop.

In terms of testing, students are given access to public tests soon after the release of assignment specifications. These can be tested using two supplied commands, the first checks the student's current version. The second checks out the student's most recent commit and tests that. This acts as a check that the students are committing correctly (and haven't forgotten to add files) and that they committed what they thought they did.

4 Questions

Three main questions to be considered when determining a functionality mark:

1. To what extent does the code give the correct answer/results in response to valid input?
2. To what extent does the code handle bad input or bad system states gracefully?
3. What does the code do while processing? / How does the code arrive at that answer?

[2] `svn checkout` for source code and `svn log` for time stamp information.

4.1 Question 1 — How does the code behave under good conditions?

All that is required here is a means to provide prepared "good" inputs; a means to capture and examine the output and actions of the system; and a set of matched inputs and outputs to indicate the correct response.

4.2 Question 2 — How does the code behave under bad conditions?

At its most basic, this just means ensuring that your test collection checks that error messages are properly triggered. Depending on the level of the course, there may be other things which should be tested. For example, empty lines (or empty input entirely) should not cause programs to loop or terminate ungracefully. There are also failures in the environment to be considered. Checking how a program responds to an instruction to read from a non-existent file is relatively easy; forcing failure to create a file because the directory is readonly requires a little more work; inducing a system call failure due to "out of resources" requires more work.

4.3 Question 3 — What does the code do while processing?

In some assessments, there may be other considerations beyond whether the code produced the correct answer. This may include whether the code:

- used forbidden calls.
- has the correct asymptotic complexity.
- has acceptable run time.
- is "safe" (eg in terms of concurrency).
- "leaks" resources.

If the assessment required that students use particular approaches, it may be possible for students to use alternate approaches or libraries which dodge the point of the assessment or avoid the work. For example reading and writing from/to disk files instead of pipes or calling built in sort functions instead of writing their own. Simple text searches for particular strings will catch some abuses but are not guaranteed to stop the truly determined (especially if the language in use is amenable to obfuscation). However, since human markers would still be looking at the code, use of obfuscation techniques would hopefully be noticed. It is up to the individual assessor how much time should be devoted to searching for this type of abuse.

Determining asymptotic complexity by sampling could be attempted programatically, provided that worst case instances are known, but is beyond the scope of this work. More general timing runs could also be used as an assessment criteria but could much more simply be used as a proxy for "must not loop indefinitely". In that case, stopping programs which "take too long" is sufficient (and a necessary self defence measure as well).

Testing safe operation under concurrency (where this is a reasonable expectation of students) would be difficult to achieve without special tools, but a rough test may be possible by testing with number of clients/requests/actions simultaneously. It will not guarantee that the code is thread safe but it may catch some instances which aren't.

Detecting resource usage generally, may be tricky or require extra tools. However, for the specific case of memory leaks, `valgind` could be employed on a number of platforms (Valgrind developers 2014).

5 Simple Automation

For this discussion, we are assuming that the following capabilities are available in some form:

1. A means to extract student submissions and compile them (where required).
2. A means to execute submitted programs programatically and to specify inputs fed to those programs.
3. A means to capture output from the executing programs.
4. A means to compare text or the contents of files with other files.
5. A means to report the results of the above.
6. A means to gather the above into a command or batch.

For example, the first item depends on the submission system, but the rest of the above can be done relatively easily with simple shell scripting or using Python's subprocess module (Python developers 2014*a*). Additional primitives which may be "nice to have" but not required include:

7. A mechanism to compare prefixes of files (eg the first 200 bytes) rather than whole files.
8. A means to automatically terminate programs which run for more than a specified number of seconds.

In more advanced settings where a number of programs may need to run simultaneously, Item 2 may require that this task doesn't block. It will also be desirable to ensure that all the started programs terminate at the conclusion of the test.

Work by Isaacson & Scott (1989) gives some examples of simple shell operations which may be useful here and also an example script. However, that script may be more detailed than is required for simple testing and would need to be customised.

With those primitives in place, the workflow for an assignment will look something like:

Pre-submission

After coming up with a concept and an initial specification:

1. *Write a working "reference" implementation of the assignment.* This allows problem areas or tasks that are harder than intended, to be identified *before* they stress the students unnecessarily. It also means that a sample solution will be available later without relying on the student body to produce one.
2. *Refine the specification (and implementation) to fix problems as they are discovered.*
3. *Create the private test set.*
4. *Create the public test set.* The "expected" outputs for both sets should be generated from the reference implementation to ensure consistency.
5. *Release assignment specification and public tests.*
6. *Update the specification and public tests.* This will be necessary if ambiguities or errors are discovered in either. To avoid problems discussed later in Section 6, it is a good idea to state that the specification trumps public tests (but that students should report contradictions so they can be fixed). If changes are made, it is important to correct the reference implementation and the private tests at the same time.

Marking

1. *Gather assignment submissions*
   The details will depend on the submission system, but a collection of subdirectories (one per student is ideal).
2. *Filter pass*
   Search for forbidden calls or commands. This is an opportunity to check the assignments for anything really nasty before compilation. See Section 8 for possible considerations.
3. *Compile submissions*
   This is quicker where the compiler has a command line interface[3]. If a build management tool such as `make` or `scons` is available then having the students submit the relevant files[4] may be helpful. Submissions which do not compile can be removed from consideration or repaired (depending on the rules of the course) at this stage.
4. *Run tests for each student*
   It will probably be necessary to monitor this process in order to restart it if one of the programs hangs[5] or kills the tester (in the case of systems programming assignments). In the case of trouble, testing can be resumed with the next submission. The problematic submission can be separated out for more cautious testing.

   In the author's experience, only a small fraction of assignment submissions ever cause problems which require manual intervention.
5. *Collate test results.*
   This is significantly easier if the per-student script/batch outputs something like a comma separated list of results (and an id) which can be concatenated and loaded into a spreadsheet for easy viewing.

   As well as being necessary for determining a grade, this can serve as a sanity check for tests. If very few submissions pass a given test, it should be reviewed to ensure the "expected answer" is correct.
6. *Rerun tests if required.*

[3]Some suites such as Visual Studio have a command line interfaces as well, but they are not always immediately obvious.

[4]Or for simple projects with known files and structure, copying a standard build file into the directory.

[5]This will only be a problem if you don't have timeouts in place.

7. *Make results and tests available to students*
   It is important to note that this does not mean that all mark components must be released at the same time. The results of automated testing can be released well before the human marked components are finished. This means that students can have an idea about how they performed quickly.

   The trick here is to find a way to make the information available in a human readable way. Adding forty columns (one for each test) per assignment to a coursework management system's marks return feature does not produce particularly readable results. An alternative would be to just make the private tests available at this point. This is not the same as the students knowing exactly what the marker recorded though. In the author's course, a simple additional program makes this fine grained information available to the students.

8. *Complete remainder of marking*
   That is, the non-automated parts.

6 Impacts/Challenges — Students

Employing this type of approach can have an impact on students. In the author's experience, three ways students can be affected (positively or negatively) are:

- There can be a collision between a strict application of a specification, and the expectation among (some) students that specifications are merely "advisory".
- Students are exposed to methodical testing and the idea of test driven development.
- Students can work "to the tests" rather than the specification.

6.1 "Advisory" versus strict specifications

Some students seem to take the view that results which vaguely match the specification are sufficient. If the students are accustomed to vague rubrics, encountering something requiring strict compliance can be a shock. While looking approximately correct may fool human markers, who have strictly limited reserves of time and alertness; the same can not be said for machine checking.

On the other hand, it may be that human markers decide to take a flexible view of matching. It is tricky though, to describe programatically the wide variety of answers which a human would consider "close enough" (eg using regular expressions). Doesn't this indicate a weakness with automated marking in that it lacks the required flexibility? Not necessarily. In many cases it is easier to specify that something should be "exactly this" instead of "something like this". It may also be that following the requirements exactly, takes no more coding effort than following them approximately.

Trying to help students by allowing greater flexibility can be counter-productive, since it often leads to students wanting a formal specification of precisely what variance is permitted and what is not. However, one way to allow some flexibility without very complex specification is to define acceptable behaviour in terms of the behaviour of standard functions and tools. For example, "if `scanf` can get the correct integer from it, then it is valid input".

Another way the warped view of the importance of following specifications manifests is in students substituting their own measures of partial success. Deimel & Pozefsky (1979) argued that "programs have to do more than just work" but the *work* aspect seems to have been deprecated. Now, students adopt measures like "hours spent" or "having written lots of code" as substitutes for doing what the specification says. This situation is certainly not unique to situations of automated marking, but it definitely occurs here.

6.2 Methodical testing

Automated marking emphasises the importance of testing for students because they are told that their marks depend on being able to produce exact matches. Some students may not have seen how effective disciplined testing can be in identifying flaws and regressions.

To allay concerns about strictness of testing, batches of public tests can be provided to students prior to the assessment deadlines. If the test mechanism is exposed to students as well, then there are additional benefits.

- The sufficiently keen students can create their own test batches and share them with fellow students.
- It is easier for students to reproduce the circumstances of a failing test in order to debug their code.
- Formative feedback and transparency: After marking, students can reproduce the marking process in order to check their marks or understand where they went wrong.
- The test mechanism can be used as an example program (especially if programs which interact with other programs are discussed in the course).

6.3 Tests versus specification

In some instances, students misuse the public tests by replacing the goal of writing a program which complies with the specification with a "simpler" goal of writing a program which passes the tests. Isaacson & Scott (1989) note that this can discourage students from considering for themselves what test inputs would be appropriate to confirm the correctness of the program.

Aside from thwarting the educational purpose of the assessment, this reliance solely on public tests is flawed on two counts: First, it can result in trying to debug a program without understanding what it is supposed to do and why. This in turn increases the risk of regressions. Second, the public tests and the hidden tests are different. Code which produces correct responses to one set without properly implementing the underlying functionality has no guarantee of doing well against a different set. Even when these facts are made known to students, the wrong emphasis seems hard to shift.

7 Impacts/Challenges — Assessment Design

As well as having impact on students, applying automation has impacts on assessment design as well.

The first consideration if automation is to used is whether the assigned task is amenable to black box testing at all. In work done in the context of programming competitions (but applicable here), Forišek

(2006) describes come features which make a task unsuitable.

- The set of possible correct answers is (relatively) large.
- Only a small amount of output is required — and that small amount of output is statistically likely to be correct.
- There is a simple but incorrect heuristic for the problem.

For example, in combinatorial problems, a program could pick an answer at random and have a non-trivial chance of getting marks. Where programs need to produce more output (which must all be coherent), this will likely be less of a problem. The last point however, has wider applicability since it is roughly equivalent to avoiding the work as described in 4.3.

Assuming that the task admits automated marking, the following factors need to be considered when describing the task and choosing what to assign marks to:

1. Precision
2. Visibility
3. Isolation
4. Determinism
5. Recognition of partial success

7.1 Precision

If something is not described sufficiently precisely and unambiguously, then it can't be tested effectively. For example, consider a program where the communication protocol between client and server for networking or IPC assessments is left for students to design; while the interface to the client (and possibly the server) is specified. This may well be desirable in more advanced assessments where students are expected to be able to do such things. However, it does mean that the network/IPC can't be tested in isolation and that both components are required to function in order for marks to be awarded. Depending on the difficulty of the task this may or may not be acceptable.

If components are to be tested separately, then a reference implementation or test rig simulating the corresponding component will be required.

7.2 Visibility

In contrast to a human marking code by inspection, with automated testing, if an event is not visible then it can't be assigned marks. Intermediate steps needed to produce results may not naturally produce output. For example, opening a network connection or successfully reading data from a file. Depending on the complexity of the overall task, it may be desirable to allocate some marks to these steps.

Actions which interact with the system (kernel) state may be visible with the right tools[6] but they seem to be either unreliable for short lived events or not simple to employ[7]. A rough test could be to produce output when each stage of the process is performed successfully. Merely outputting "Success" does not mean it actually happened though, so the code would also need to be tested to see if it accurately reports failures. Practically, it seems fairer to explicitly test for failures and leave successes to be assessed in later steps.

In the case of intermediate results, there may be ways to expose them but that exposure must also be specified. This may enlarge/complicate the assessment specification further. The presence of extra internal state information may distort the output of the program with clutter. It may also give students an unrealistic view of programming practice.

[6]Possibilities include: polling with system tools or library interposing.

[7]This is not to say that they lack merit, merely that they fall outside the scope of "simple automation".

7.3 Isolation

If an event or result can't be isolated from other output, then it can't be marked. If a result is indicated by the presence of an easily extracted string in the output, then this may not present much of a challenge. However, if the test is equivalent to "is the first part of the output is correct?", it is a bit more fiddly. Two solutions here are either:

1. compare only prefixes of the output and ignore any differences after a certain threshold.
2. Ensure that the program/system can be stopped as soon as possible after the event of interest.

The first option is not difficult to code but we want to minimise any custom coding required so let's consider the second one. A simple way to have natural stopping points is to have programs which process distinct operations and prompt between them. Then specify what should happen at end of input / disconnection. This means that the most basic unit of simple testing is "empty input"[8]. More sophisticated tests can then be built up from that starting point: one interaction then stop; two interactions then stop, ...

The importance of handling end of input properly should be emphasised to students, but if public tests are available, then failures in this aspect will be readily apparent. There is a side benefit here in that handling end of input properly is not something which students seem to consider when left to their own devices.

7.4 Determinism

To keep testing simple and transparent, the behaviour of (correctly written) programs being tested should be deterministic. This does limit the use of things like random numbers unless a pseudo-random number generator is specified and the seed can be easily specified. Rather than do this, a simpler option is to read streams of values from files rather than the randomiser. For example, instead of shuffling cards, specify a file which contains a pre-ordered deck.

Where a number of processes or threads are involved, accidents of scheduling can lead to race conditions. Two cases to consider here:

1. The ordering/interleaving of output varies, but the decisions made by the programs are the same. If success can be determined by the presence of particular strings in the output, then a search could be made just for those strings. If the output from different parties can be distinguished somehow (eg relevant lines have a known prefix), then the relevant lines can be filtered. Alternatively, simply sorting the lines of text before comparison deals with the ordering issue quite neatly[9].

[8]After argument checking to allow the program to start at all.

[9]Assuming that any ordering is equally valid

2. The programs make different decisions under different event orderings. This occurs in situations like networking assignments where a number of clients need to start and connect to a server. For example, the clients represent players in a game and which "seat" the player occupies is significant. Introducing pauses after starting the server and between each client start would seem to be a solution here. However,

    - determining the time to wait can be tricky. Too long a delay and the tests will take a long time to run, frustrate users and slow down the testing unnecessarily. If the delay is too short, the tests may react erratically on a heavily loaded machine.
    - forcing the testing to operate in serial, removes the need for students to write thread-safe code.

    An alternative solution is to make the ordering depend on something predictable. For example, requiring each player to give their name and then seating players in lexicographic order, gives predictable results[10] but does not force connections to be spaced out.

7.5 Recognition of partial success

Testing for matching output does not leave much room for "part marks". Either the program passes the test or it doesn't. This can mean that a program which has 90% of a task perfectly correct could still record a "fail" for that task. Alternatively, a program could behave correctly under most but not all correct inputs. To mitigate this risk:

- A number of tests should be employed to mark against each subtask to help recognise programs which are capable of completing that task under some conditions.
- The tasks for the assignment should be examined in the light of precision, visibility and isolation to see whether they *should* be subdivided. This may require interface changes, such as extra output.

The subdivision of tasks needs to be appropriate for the level of the course. A balance needs to be struck here between rewarding partial progress versus the need for programmers to produce working code.

8 Security and Integrity Considerations

The preceding discussion assumes that the marker runs student code or build instructions in a context other than the student's account. This will always be risky to some degree: Programs could be submitted which attempt to gain access to system privileges or to course information; or to disrupt the marking process itself. Alternatively, programs may simply be badly written and abuse system resources.

Possible approaches to *mitigate* risks could be some combination of:

- *Identifying problem programs before running them.* This will never be possible in the completely general case but simple checks can be made for calls to external programs. Depending on the programming language, any uses of inline assembly language or unusual compiler directives are candidates for further examination.
- *Preventing programs from being able to do undesirable things.* Approaches such as library interposing or automated code substitution to replace "potentially dangerous" calls with more restricted ones. The author has employed this in the past to protect against fork bombs. In the general case, this would be harder because all of the valid ways a call could be used would need to be accounted for.
- *Isolating "bad" code so there is nothing for it to attack.* This could include running code in a separate/limited account but could extend to running in a separate environment. For example, a virtual machine or a chroot/jail. Steps in this category may require help from systems administrators.

Care needs to be taken however that the environment used for marking does not differ significantly from the one accessible to students. If the test environment is protected in ways that cause it to behave significantly differently to the students' environment under normal conditions, then questions of fairness must be considered. On the other hand, if the development environment is "too safe", students will not learn how to recognise and debug problems "in the wild". With this in mind, where possible, protection measures should be optional in that they can be applied (or not) without affecting the main results. See (Ihantola et al. 2010) for other possibilities.

9 Conclusion — Where is the work hiding?

Now that we have these primitives and workflow for assignments which will be marked programatically, we now summarise the important question of "where is the work hiding?". After all, the fact that simple tests can be administered programatically, does not make the process trivial.

Much of the work in dealing with this type of assignment is front-loaded. The specification, reference implementation[11] and public tests are all needed before the assignment is released. Decisions about precisely how programs will be evaluated can't be deferred until a time after the assignments have been submitted. This work requires a greater amount of the teacher's time, with reduced amount of tutor/teaching assistant time. The net budgetary affect of this shift needs to be considered.

Significant detail is required in the specification. It must describe precisely how the program is to behave and what the output and side effects are to be. Examples of interactions will probably be needed. All this means that while specifications may not actually be more complicated, they may be large. In the author's case for a second year course, this results in specifications of (roughly) between five and eight pages once common boiler-plate text is removed.

In conclusion, while simple automation (however it is accomplished) does move some work earlier in the course, it can significantly reduce the marking burden. Further, because of the reduced academic effort involved when dealing with submissions, scales quite well.

References

Benson, M. (1985), 'Machine assisted marking of programming assignments', *SIGCSE Bull.* **17**(3), 24–

[10]Yes, this assumes one agrees not to test with duplicate identifiers.

[11]While it is possible to start with a partially complete implementation, doing so creates problems later.

25.
URL: *http://doi.acm.org/10.1145/382208.382516*

Blumenstein, M., Green, S., Fogelman, S., Nguyen, A. & Muthukkumarasamy, V. (2008), 'Performance analysis of game: A generic automated marking environment', *Computers and Education* **50**(4), 1203–1216.

Deimel, Jr., L. E. & Clarkson, B. A. (1978), The todisk-watload system: A convenient tool for evaluating student programs, *in* 'Proceedings of the 16th Annual Southeast Regional Conference', ACM-SE 16, ACM, New York, NY, USA, pp. 168–171.
URL: *http://doi.acm.org/10.1145/503643.503681*

Deimel, Jr., L. E. & Pozefsky, M. (1979), 'Requirements for student programs in the undergraduate computer science curriculum: How much is enough?', *SIGCSE Bull.* **11**(1), 14–17.
URL: *http://doi.acm.org/10.1145/953030.809543*

Douce, C., Livingstone, D. & Orwell, J. (2005), 'Automatic test-based assessment of programming: A review', *J. Educ. Resour. Comput.* **5**(3).
URL: *http://doi.acm.org/10.1145/1163405.1163409*

Edwards, S. H. (2003), Teaching software testing: Automatic grading meets test-first coding, *in* 'Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications', OOPSLA '03, ACM, New York, NY, USA, pp. 318–319.
URL: *http://doi.acm.org/10.1145/949344.949431*

Forišek, M. (2006), 'On the suitability of programming tasks for automated evaluation', *Informatics in Education* **5**(1), 63–73. Copyright - Copyright Institute of Mathematics and Informatics 2006; Document feature - ; Last updated - 2011-06-03.

Forsythe, G. E. & Wirth, N. (1965), 'Automatic grading programs', *Commun. ACM* **8**(5), 275–278.
URL: *http://doi.acm.org/10.1145/364914.364937*

Fowler, M. (2014), 'Xunit'.
URL: *http://www.martinfowler.com/bliki/Xunit.html*

Harris, J. A., Adams, E. S. & Harris, N. L. (2004), 'Making program grading easier: But not totally automatic', *J. Comput. Sci. Coll.* **20**(1), 248–261.
URL: *http://dl.acm.org/citation.cfm?id=1040231.1040264*

Hollingsworth, J. (1960), 'Automatic graders for programming classes', *Commun. ACM* **3**(10), 528–529.
URL: *http://doi.acm.org/10.1145/367415.367422*

Ihantola, P., Ahoniemi, T., Karavirta, V. & Seppälä, O. (2010), Review of recent systems for automatic assessment of programming assignments, *in* 'Proceedings of the 10th Koli Calling International Conference on Computing Education Research', Koli Calling '10, ACM, New York, NY, USA, pp. 86–93.
URL: *http://doi.acm.org/10.1145/1930464.1930480*

Isaacson, P. C. & Scott, T. A. (1989), 'Automating the execution of student programs', *SIGCSE Bull.* **21**(2), 15–22.
URL: *http://doi.acm.org/10.1145/65738.65741*

Joy, M., Griffiths, N. & Boyatt, R. (2005), 'The boss online submission and assessment system', *J. Educ. Resour. Comput.* **5**(3).
URL: *http://doi.acm.org/10.1145/1163405.1163407*

JUnit project (2014), 'JUnit FAQ'.
URL: *https://github.com/junit-team/junit/wiki/FAQ*

Python developers (2014*a*), '17.1. subprocess Subprocess management Python v2.7.7 documentation'.
URL: *https://docs.python.org/2/library/subprocess.html*

Python developers (2014*b*), '25.3. unittest — Unit testing framework — Python v2.7.7 documentation'.
URL: *https://docs.python.org/2/library/unittest.html*

Solomon, A., Santamaria, D. & Lister, R. (2006), Automated testing of unix command-line and scripting skills, *in* 'Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th International Conference on', pp. 120–125.

Valgrind developers (2014), 'Valgrind: Supported platforms'.
URL: *http://valgrind.org/info/platforms.html*

What Are We Doing When We Assess Programming?

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

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

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

Abstract

Considerable research has been devoted in recent decades to identifying optimal pedagogical strategies for teaching computer programming. Often the result of these efforts has been to conclude that we have made little progress, as our students consistently perform poorly on the assessments we apply to measure teaching efficacy. In this paper, we suggest that a significant contributor to this poor performance may be the methods of assessment, which do not reflect the knowledge and skills that a real programmer needs to write real code. We propose an alternative assessment format using Activity Diagrams that better reflects true programming ability. Our preliminary results indicate that these assessments correlate well with the ability to produce working code, while more traditional question formats do not.

Keywords: Programming Education, Programming Assessment.

1 Introduction

In her seminal 1999 paper, Sally Fincher asked "What are we doing when we teach programming?" (Fincher, 1999) Fincher noted that there had been a shift in the perception of computer programming from a mechanical skill to an essential element of computer science theory, without a consensus on the implications of this shift for pedagogy. This discussion helped to launch CS Education as a formal area of academic research.

In the intervening 15 years there has been extensive exploration of approaches to the teaching of programming, comparing languages, tools and conceptual methodologies (e.g. Sajaniemi, Kuittinen and Tikansalo, 2008; Pears, Seidman, Malmi, et al., 2007).

Unfortunately, the most consistent conclusion drawn from this body of research is that in spite of all this effort we still don't know how to teach programming, because large numbers of our students fail our introductory programming courses (Bergin and Reilly, 2005; Gonzalez, 2006; Lahtinen, Ala-Mutka and Järvinen, 2005) and even those who pass don't seem to be able to program very well (Ford and Venema, 2010; Thomas, Ratcliffe, Woodbury, et al., 2002; Bornat, Dehnadi and Simon, 2008).

It seems illogical that after so much study, and after having produced a sufficient number of skilled programmers to drive the computing industry beyond recognition in the last two decades, we still conclude that our students can't program. Perhaps an alternative is possible: we're teaching successfully -- but we're assessing badly. Specifically, when we assess, we are confusing programming with the abstractions of computer science and symbolic manipulation. Students fare poorly on our assessments not because they can't program but because we are not testing their programming ability. A cow makes a terrible racehorse, but that doesn't mean she's not a very good cow.

1.1 The Role of Assessment

Accurate assessment of programming ability has multiple roles in programming education and programming education research. Primarily, of course, we wish to measure programming ability in the classroom to evaluate and rank students, deciding who passes and who does not pass our programming courses. In programming education research, we also use measures of programming ability as our dependent variables. We may, for example, wish to evaluate a teaching intervention by comparing students' programming ability with and without the technique. This requires an accurate measure of individual programming ability.

Often these two roles overlap. For example, in studies of teaching efficacy, researchers often use final course mark as a reflection of programming ability. Cardell-Oliver has clearly articulated the potential weakness of this approach, (Cardell-Oliver, 2011) but it remains extremely common (cf. Clear, 2008).

Thus our ability to accurately measure an individual's programming skill underpins both our educational and scientific endeavours.

The critical nature of assessment in both of these roles -- teaching and research -- is recognised by the CS Education community. Simon, Sheard, and their colleagues (e.g. Sheard, 2012; Simon, Sheard, Carbone, et al., 2012) have undertaken exhaustive descriptive studies of the types of questions used in examinations in programming courses. Various authors have carefully mapped individual exam questions to the Bloom and SOLO taxonomies (e.g. Lister, Simon, Thompson, et al., 2006; Whalley, Lister, Thompson, et al., 2006) in order to determine more precisely what is being tested by programming exams.

Discussion of metrics in research is also vigorous. Considerable debate has touched on whether McKracken's seminal "our students can't program" study (McCracken, Almstrum, Diaz, et al. 2001) was biased by the use of an invalid metric of programming ability (Utting, Tew, McCracken, et al., 2013; McCartney, Boustedt, Eckerdal, et al., 2013). An assortment of tools have been proposed, explored and validated for use in research studies of programming education.

Study of this body of research gives clear insight into ***how*** we are assessing. It is our contention, however, that further analysis is required into ***what*** we are assessing.

1.2 Programming or Computer Science?

Fincher (Fincher, 1999) originally made the distinction between programming as a means to an end, and programming as a component of a theoretical discipline. She noted that before Computer Science existed as the unique theoretical discipline underpinning computation and computation systems, practitioners of engineering, chemistry and mathematics used programming as a way "to get the computer to do something". As Computer Science matured, programming became an area for theoretical exploration in its own right in the contexts of automata theory, language design, and compiler construction. It remained, however, the essential tool for getting computers *to do things*. Since 1999, computers and computing have become so ubiquitous that in nearly every academic and commercial discipline, getting computers to do things is indispensable. Not only the software development industry, but many associated disciplines that rely on computer software require people with advanced abilities in, specifically, computer programming. We contend then, that Fincher's maturation process has come full circle -- we now have two fully mature, separate but intertwined disciplines: computer programming and theoretical computer science. Computer programming is the ability to produce working digital artefacts to the standards dictated by industrial best practice. Computer science is the study of the underlying principles of computing and computation.

It is possible to find tertiary degree programmes that quite clearly direct their students in one or the other of these disciplines. One group seeks to produce graduates who are prepared to step into the information technology industry as software developers versed in the skills and techniques required of a professional programmer. The other seeks to prepare students for further study in experimental and theoretical areas of computer science. Both of these disciplines are relevant and challenging and, while they share many fundamentals, they clearly place different emphasis on the elements of computer science.

We believe that in many attempts to assess programming ability, both as an educational evaluation and as a performance metric in quantitative research, these two disciplines are becoming entangled. For example, Lister and his colleagues (Lister, Adams, Fitzgerald, et al., 2004) describe a suite of multiple choice questions that can be used to evaluate a student's ability to read code, an essential part of the ability to program (i.e. to generate working digital artefacts with a programming language). Among the questions is the following code fragment:

```
int[ ] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;

while (i < j)
{
        temp = x[i];
        [i] = x[j];
        [j] = 2*temp;
        i++;
        j--;
}
```

Students are asked to identify the contents of the array x after these statements are executed. This code fragment is written in syntactically correct Java and it is semantically coherent, but it isn't a piece of code that a professional programmer would ever write. It performs no logically delineated task; it provides no context for the purpose of the computation. Not being able to answer this question doesn't necessarily demonstrate that one is unable to write code to iterate over and modify the contents of an array, it merely indicates that this abstract and tortuous piece of code is difficult to understand.

Similarly, Dehnadi (Dehnadi, 2006) presents a suite of questions to test understanding of the concept of assignment in programming. Among the Dehnadi questions is this one:

```
int a = 5;
int b = 3;
int c = 7;
a = c;
b = a;
c = b;
```

The student is asked the values of a, b, and c after execution of these statements. Again, this is syntactically correct code, and it is certainly possible to work out the values of the variables. But it is very difficult to come up with a realistic scenario under which a programmer would actually write this fragment.

Questions such as these therefore attempt to test the ability to perform the act of programming by requiring an understanding of something that would never be done while performing that act. Students traditionally score very badly on these questions (the question from Lister had only a 73% success rate, even though it was a 5-item multiple choice, and this was actually the highest success rate of the entire suite). Under the assumption that these questions are testing the ability to program, we conclude, dispiritedly, that our students cannot program. However, if we accept that the ability to program is the ability to write digital artefacts to an industrial standard, this pessimism may be unwarranted. Since no one would need to produce such code to be a capable programmer, our students' failure to correctly answer these questions does not mean that they cannot program. It simply means that they cannot do whatever it is that these questions require.

This disconnect can be seen in authors' descriptions of their measurement tools. For example, Ford and

Venema's oft-cited work on assessment (Ford and Venema, 2010) is entitled "Assessing the Success of an Introductory Programming Course". However, the authors state quite specifically that they are not trying to assess the ability to program as we have defined it (i.e. the ability to produce working digital artefacts following current standards of best practice). Instead, they propose a suite of tests "to examine whether students who have passed an introductory course have achieved an understanding of fundamental concepts in programming" (pg. 1). That is, they wish to measure the mastery of fundamental concepts, rather than the specific skill of programming. Note that the former is not sufficient to demonstrate the latter -- that is, I may have a detailed understanding of the physical principles of flying, but still not be able to safely launch myself into the air. One might argue that mastery of fundamental concepts is at least necessary to demonstrate the skill of programming, but this presupposes that what has been identified as fundamental concepts are, in fact, fundamental and, more critically in this context, that they are being tested in ways that are relevant to the skill we are attempting to quantify.

Ford and Venema focus specifically on the concepts of "assignment and sequencing". These are clearly fundamental to the act of programming because modern programming languages provide constructs for both assignment and sequencing and all digital artefacts of nontrivial complexity must include these elements. Ford and Venema explore a number of existing metrics which test comprehension of the principles of assignment and sequencing, but which do so at a very high level of abstraction. For example, in addition to the questions of Dehnadi described above, and a similar suite from Ma (Ma, Ferguson, Roper, et al., 2007), they include the "Reges question":

> *If b is a Boolean variable, then the statement*
> *b = (b = = false);*
> *has what effect?*

This item certainly requires an understanding of the rules of assignment and sequencing, but it is not typical of the constructs working programmers ordinarily produce and thus, we suggest, does not pragmatically test the ability to program.

Ford and Venema observed generally poor performance on all of their metrics and concluded "that many students who had passed an introductory programming course had little or no understanding of fundamental concepts" (pg. 1). We would suggest that this conclusion does not necessarily follow. What one can conclude from the poor performance on such tools is that many students who had passed an introductory programming course were unable to solve these complex and abstract problems involving the assignment operation. They might still understand the fundamental concepts of programming, and perhaps, be able to program quite adequately for their level of experience. Being able to solve complex, abstract symbolic manipulation exercises is an extremely valuable skill, and one a person might certainly wish, in some circumstances, to be able to measure, but it is not equivalent to the skill of programming.

1.3 How to Measure the Ability to Program

We have argued that often, the traditional pencil-and-paper approaches to programming assessment (particularly code reading problems) are presented at a level of abstraction that makes them poor indicators of the ability to actually write good, working code. We need an alternative method to assess programming ability. The obvious suggestion is to have students create digital artefacts and assess them for functional accuracy and code quality via industrially approved metrics. We do, in fact, contend that this should be the Gold Standard for judging one's ability to program. Unfortunately, in the practical contexts of both student evaluation and educational research, this approach has its own significant shortcomings.

First, in many situations, using artefacts for assessment is prohibitively expensive. In large classes, it may simply be impossible to hand mark the large number of pieces of code that would be required. Even in our own institution, where courses are limited to no more than 48 students per semester, marking of coding assignments is a huge burden on teaching staff. Promising work is being done in the area of automated assessment of code (Ihantola, Ahoniemi and Karavirta, 2010), but it is not a problem that has yet been completely resolved. The same objection would apply to any research study that wished to incorporate artefact assessment as a measure of efficacy.

Second, in the classroom situation, issues of authorship arise when assessment is performed outside of controlled examination conditions. Many educators have noted concerns about plagiarism in student coding projects. While there are some tools that can be used to help detect plagiarism (e.g. Vamplew and Dermoudy, 2005) it again makes artefact marking a questionable choice, at least as the sole means of evaluation.

Third, artefact evaluation is impractical until actual artefacts can be produced, that is, until a certain level of coding skill has been attained. In the face of increasingly compelling evidence for the importance of effective pedagogy and the detection of difficulties in the very earliest weeks of programming education (Robins, 2010), it would be very risky to delay evaluation until students are experienced enough to write substantial pieces of software.

There are two steps that can be taken to ameliorate these difficulties. Most obviously, in a classroom situation, one can compose a course mark from a mixture of code artefacts and assessments performed under exam conditions. In our own introductory programming courses, we use a combination of code projects developed outside of class, written examination, and practical coding exercises conducted under examination conditions.

Further, when conducting written examinations, one can attempt to construct questions which reflect, as accurately as possible, skills used by *real programmers* in writing *real code*. To this end, it can be helpful to refine our notion of the fundamental concepts and capabilities of computer programming.

1.4 Programming Fundamentals

We have argued for computer programming as a profession distinguished from theoretical computer science, based on the ubiquity in the modern world of computer artefacts and computation. A related argument is increasingly made for not only the act of programming, but the act of "thinking like a programmer." The term generally used in this discussion is "computational thinking" (Wing, 2006). Although there is some variability amongst authors, the basic tenet of the computational thinking movement is that computers are used as aids to problem-solving in most professions, activities and endeavours in modern Western society. Further, that there is a common underlying style for solving problems with a computer based on problem decomposition and algorithmic construction -- effectively translating our human solutions into something a computer can do. It has been argued that being able to perform this style of problem solving, i.e. being able to think computationally, is as integral in modern society as literacy and numeracy. Computer programming -- by definition, getting a computer to do things -- naturally exhibits the computational thinking template. In this context, we view the process of producing a computer program to solve a specific problem as being comprised of three steps:

1. Conceptualising a solution to the problem in the domain of human cognition (the way a human would do it).
2. Symbolic translation of the human solution into computer operations (the way a computer could do it).
3. Concrete translation of the computational solution into a specific programming language (the generation of running code).

For example, if one wishes to implement a greedy algorithm for the Knapsack problem, the human solution would be expressed as "pick the highest value item that will fit"; the computer's solution would be something akin to "sort all items descending by value; iterate over the sorted list; compare each item's size to that remaining in the knapsack; choose the first one that fits". Note the much more decomposed and sequential nature of the computer's solution. Finally, the computer's solution would need to be written out in whatever programming language was being used, incorporating the particular syntactic rules and features of that language.

These three steps all involve both comprehension and production. For example, a programmer must be able to produce the computer's solution in step 2, and must be able to understand the logic of a computer's solution in order to translate it into code for step 3.

After identifying these three core activities of programming, we can then attempt to develop assessment tools that require these activities to be performed. In the next section, we present the technique we are currently exploring, and some preliminary evidence of its accuracy.

1.5 Assessment with Activity Diagrams

In recent offerings of our first programming course, we have begun using Activity Diagrams (e.g. Schmuller, 2004) as both a teaching and an assessment tool. Our preliminary data indicate that performance on these problems may correlate better with our Gold Standard (evaluation of artefacts) than do more traditional multiple-choice code reading questions.

Activity Diagrams are a component of the UML methodology for software development (e.g. Schmuller, 2004). They are used to diagram spatially the logical flow of a computer program, and have a notation for sequence, conditionals and looping. They do not depend on the syntactic details of any particular programming language. For our novice students, we use a simplified version of the full UML technique that eliminates some elements and requirements that are extraneous to the code written in a first programming course. An example for a program to play a simple "card game" is shown in Figure 1.

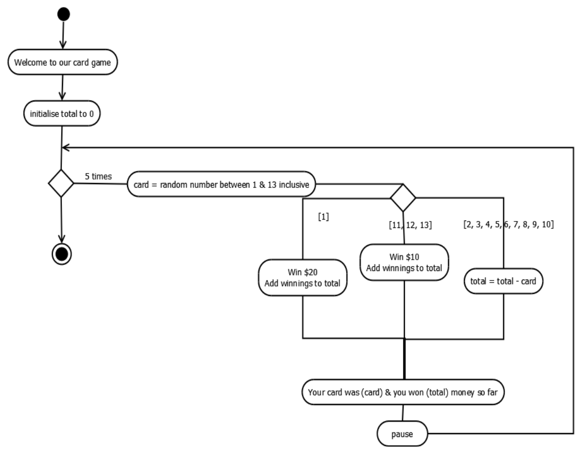

Welcome to our card game
initialise total to 0
5 times
card = random number between 1 & 13 inclusive
[1]
[11, 12, 13]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
Win $20 Add winnings to total
Win $10 Add winnings to total
total = total - card
Your card was (card) & you won (total) money so far
pause

Figure 1: Activity Diagram for Simple Card Game Program

Activity Diagrams are similar to the traditional flowchart, which has been used to represent code structure for decades. In the early years of computer programming, the flowchart was studied extensively as a tool for program documentation, with mixed, sometimes controversial, results. After their introduction in the late 1940s (Goldstein and von Neumann, 1947) flowcharts rose rapidly in popularity to become an expected part of every programmer's skill set (cf. Schneiderman, Mayer, McKay et al., 1977). Flowcharts in these decades were used primarily as documentation tools. That is, a flowchart was produced to illustrate the structure and logic of a piece of software. Flowcharts were also used as an educational tool to illustrate complex algorithms (cf. Scanlan, 1987). Unfortunately, human factors studies performed in the 1980s demonstrated that flowcharts provided no advantage over an actual code listing for comprehension, debugging or modification (see e.g. Shneiderman, 1982). It was suggested that modern (for the time) high-level programming languages were more useful for representing program logic than were the graphical techniques of flowcharting (Ramsey, Atwood and Van Doren, 1983).

We use Activity Diagrams not for code documentation, but as teaching tools and for both formative and summative assessment. We have found them promising in all three contexts. However, in this discussion we will consider only their role in summative assessment or, equivalently, as a measure of programming ability in a research situation.

Previously, we described the journey from problem statement to working program as involving three steps: generating the human solution; translating to the computer's solution; translating into a specific language. Because Activity Diagrams are relatively language-agnostic, they give the student an opportunity to express the first two of these steps with a reduced burden of syntactic detail. By having the student translate an Activity Diagram into a specific programming language, we can test the third step.

In assessment, by selectively controlling the role of the activity diagram, we can isolate the three steps of digital artefact generation, and can assess both code production and comprehension. For example, if we give a student a problem statement in English and ask him or her to draw the corresponding Activity Diagram, we can observe the student's ability to produce the output of steps 1 and 2 (make a human solution; translate it into something the computer can do) from the procedure outlined above. If we give the student an Activity Diagram and ask him or her to explain the purpose of the resulting code, we are testing comprehension (code reading) of the same steps. Each of the various combinations of question content and task can serve to exercise one or more aspects of the programming process.

We believe that examination questions involving Activity Diagramming measure more accurately what real programmers do than the often abstract (and sometime artificial) code reading problems discussed above. In the present study, we wished to measure the relationships between student performance on a manually marked large programming project, assessment items involving Activity Diagramming, and assessment items using more traditional multiple-choice and short answer formats. Under the assumption that the project mark is the best available measure of true programming ability, we anticipated stronger correlations between project mark and scores on the Activity Diagram problems than between project mark and scores on the traditional problems.

2 METHODOLOGY

In two recent offerings of our CS1 first programming course, students' final course marks were computed as the weighted average of a) a set of in-class practical exercises; b) an in-class programming task under exam conditions; c) a large individual programming project and d) a written theory test performed under exam conditions. The in-class practicals were performed throughout the semester and the in-class programming task was performed half-way through the semester. These components were treated as both summative and formative assessment, with the students receiving detailed feedback on their progress, and additional tutorial support for any identified difficulties.

The out-of-class programming project was assigned four weeks before the end of the semester and was due on the last day. The written exam was given in the final week of the semester. These components were used for summative assessment.

The out-of-class programming project was a simple trivia game that required file I/O, random selection, and comparison. Upon submission, each student's solution was marked by an experienced programming teacher by hand using a detailed grading rubric[1], which assesses for both correct functionality and code quality. The marker checked carefully for instances of excess similarity between pairs of assignments, and no detectable incidences of plagiarism were identified. We acknowledge that stronger protection against plagiarism would be preferable (to insure, for example, that students are not getting help from more experienced programmers), but it is not practical to simultaneously provide an opportunity for a substantial programming exercise and to rigorously observe each moment of that programming process. For the present analysis, the marks on the out-of-class project serve as each student's Gold Standard. That is, the mark on a student's project is taken to be the most accurate reflection available of his or her true ability to produce a working digital artefact at the finish of this introductory programming course.

The written theory exam[2] contained 15 questions using a variety of formats including traditional multiple-choice code reading and writing questions, short code production exercises, and one Parsons Puzzle (Parsons and Haden, 2006). In addition, in one question (Question 12), students were given a syntactically correct code sample and asked to draw the corresponding Activity Diagram. In one question (Question 13), students were given a problem statement in English and asked to draw the corresponding Activity Diagram. All exams were marked by a single, experienced programming tutor.

72 students completed both the written theory exam and the out-of-class project, and their results are included in the following analyses.

3 RESULTS

Pearson-product moment correlations (point-biserial correlations for dichotomous problems) between the out-of-class project mark and question score were computed for each of the fifteen questions on the written theory exam. The results, along with the format of each exam question, are shown in Table 1.

[1 & 2]Available from Dale.Parsons@op.ac.nz

Question	Question type	Correlation with Project Mark	p-value
Q01	MCQ code writing	0.068	ns
Q02	MCQ code reading	0.149	ns
Q03	Short answer code reading	0.133	ns
Q04	MCQ code writing	0.252	p <.05
Q05	Short answer code reading	0.143	ns
Q06	MCQ code reading	0.216	ns
Q07	MCQ code reading	0.330	p<.01
Q08	Short answer code reading	0.183	ns
Q09	MCQ code reading	0.063	ns
Q10	Problem statement -> Code	0.224	marginal p=.059
Q11	Problem statement -> Code	0.423	p<.01
Q12	Code -> Activity Diagram	0.258	p<.05
Q13	Problem Statement -> Activity Diagram	0.352	p<.01
Q14	Short answer problem statement -> Code	0.279	p<.05
Q15	Parsons Puzzle	0.118	ns

Table 1: Correlations between exam questions and out-of-class project mark

Significant positive correlations with out-of-class project mark were found for two of the nine traditional MCQ or short answer questions (Questions 4 and 7). All three problems that required code writing were significantly (Questions 11 and 14) or marginally significantly (Question 10) correlated with project mark. Both Activity Diagram questions (Questions 12 and 13) were significantly correlated with project mark. All other observed correlations were non-significant.

The observed correlation between performance on the code writing questions (10, 11 & 14) and earned mark on the major code writing assignment is to be expected. One would assume that students who can write code well out of class are more likely to write code well on an exam. This correlation provides encouraging support for the validity of project mark as our "Gold Standard" of programming ability. It should be noted, however, that explicit code writing questions may be unable to discriminate between the three steps of the programming process proposed above. A student's failure to successfully write code on an exam might be due to difficulty translating his or her solution into computer operations, or due to difficulty expressing computer operations in syntactically correct programming language, or to some combination of the two. The complexity of explicit code writing exercises must also be severely restricted in the earliest weeks of a programming course, where syntactic mastery has not been achieved.

Among the best predictors of out-of-class project mark were the two questions where students were given a code sample or a problem statement and asked to generate an Activity Diagram.

This type of problem maps directly to our posited process for generating working code. It captures a student’s ability to perform critical computational thinking, with a reduced burden on syntactic accuracy. It tests this aspect of their ability to program in a way that is neither artificially complex nor artificially abstract. And, at least for this sample, it correlates significantly with the ability to produce a nontrivial working application (i.e. the ability to program).

While it is of interest to note that performance on the Activity Diagram questions is significantly correlated with the score on the out-of-class project, it is equally interesting to note that performance on the majority of the traditional format questions is not. For example, question 2 is a standard multiple-choice code reading question:

What are the values of girls, boys, and children after the following code has been executed?

```
int girls = 0;
int boys = 0;
int children = 0;
children = girls + boys;
girls = 15;
boys = 12;
```

(a) 0, 0, 0
(b) 0, 0, 27
(c) 15, 12, 0
(d) 15, 12, 27

The observed correlation between marks on Question 2 and our Gold Standard measure of programming ability is small -- only 0.149 -- and nonsignificant.

Non-significant correlations between performance on a test item and mark on the out-of-class project can be caused by a variety of numerical patterns. Most obviously, a low correlation occurs when success on the test item is not consistently associated with a high mark on the project, and symmetrically, failure on the test item is not consistently associated with a low mark on the project. However, a low correlation will also occur if the test item is so easy that most students get it right (ceiling effect) or so difficult that most students get it wrong (floor effect). In this case, performance on the item is largely the same for all students regardless of performance on the project, and the Pearson-product moment correlation will be near zero. In either of these scenarios, score on the test item is not a sensitive measure of programming ability.

A summary of performance for each exam question is shown in Table 2. The nine traditional format questions (Q1 to Q9) are graded "all or nothing", that is, no partial credit is given. For those items, Table 2 presents the percent of correct responses across all students. Note that the multiple-choice questions (numbers 1, 2, 4, 6, 7 & 9) each provide four response alternatives, so one would expect a 25% correct response rate simply due to chance. Problems 10 to 15 are marked out of a fixed number of points, and partial credit is given. For those problems, Table 2 shows the average proportion of available marks earned, across all students.

Question	Question type	Percent Correct	Average Pr(Marks)
Q01	MCQ code writing	0.81	
Q02	MCQ code reading	0.78	
Q03	Short answer code reading	0.93	
Q04	MCQ code writing	0.81	
Q05	Short answer code reading	0.71	
Q06	MCQ code reading	0.60	
Q07	MCQ code reading	0.47	
Q08	Short answer code reading	0.71	
Q09	MCQ code reading	0.86	
Q10	Problem statement -> Code		.76
Q11	Problem statement -> Code		.66
Q12	Code -> Activity Diagram		.79
Q13	Problem Statement -> Activity Diagram		.40
Q14	Short answer problem statement -> Code		.59
Q15	Parsons Puzzle		.86

Table 2: Performance summary for all exam questions.

The values in Table 2 suggest that both sources of low correlation described above are present in our results. Question 3, for example, shows a 93% correct response rate and a non-significant correlation of 0.133 with the out-of-class project. Since most students answered this question correctly, it fails to discriminate between those students who perform well on the out-of-class project and those who do not. Question 6, in contrast, shows a moderate 60% correct response rate and a non-significant correlation of 0.216. This question does not appear to suffer from a floor or ceiling effect, so performance on it is simply not well-correlated with assignment mark. In both cases, these traditional format questions are not usefully predictive of performance on the out-of-class project.

Our Questions 1 to 9 are representative of the type often used to assess programming ability in computer science education research, including those studies which conclude that our students cannot program because of low success rates on these test items. If in fact, the ability to solve this kind of test item is not well correlated with the ability to program, we may be encouraged to posit that our students may, in fact, be able to program, but that this type of question does not accurately measure that ability.

These results are of course only preliminary. The sample size is only moderate, and one should not infer too much from individual, possibly pathological, exam questions. Nonetheless, there is a clearly detectable pattern that encourages us to believe that through the use of Activity Diagram questions, we might be able to obtain an acceptably accurate measure of true programming ability without the often prohibitive cost of hand-marking complete software artefacts.

4 DISCUSSION

In recent decades, computer programming has progressed from an isolated esoteric tool, to a critical element of a scientific discipline, to a vocational practice on which many of the world’s systems and institutions now depend. Throughout this process, programming pedagogy has struggled to achieve an always successful delivery of what is an inherently difficult subject to teach. This difficulty has led to the gloomy conclusions that we can’t teach it, and our students can’t do it.

In this paper, we have suggested that this sad situation may be due, at least in part, to the methods we use to assess programming ability, both in research and in the classroom. We propose that the types of questions often used to assess programming ability are not measuring its most essential aspect – the ability to produce good quality working code. As an alternative, we propose the use of a spatial representation of coding logic – the Activity Diagram – that may more accurately reflect what we mean when we ask “can our students program”? Our early experience with this technique indicates that it correlates better with the ability to produce quality working code than do traditional multiple choice and short answer questions.

One of the reasons that our students score so poorly on traditional written programming questions is that they are often confusing. In order to avoid trivially simple non-discriminatory assessments, we must make these questions convoluted and abstract, often to the point where they no longer represent the realistic cognitive behaviours of real programming. Activity Diagram questions, in comparison, are inherently complex. To generate an Activity Diagram, the student must generate a computationally appropriate solution to a problem, and express that solution using the Activity Diagram notation. There is enough challenge in this activity that we do not need to artificially complicate the problem to avoid triviality. We can use realistic programming contexts to assess real programming skill.

Our results, while preliminary, encourage us to hope that perhaps we haven’t really been doing such a bad job of teaching programming, although we may have been doing a questionable job of assessing it.

5 REFERENCES

Bergin, S. and Reilly, R. (2005): The influence of motivation and comfort level on learning to program. *Proceedings of the 17th Annual Workshop on the Psychology of Programming Interest Group* pp 293-304, University of Sussex, Brighton UK 29, June – 1 July, 2005.

Bornat, R., Dehnadi, S., and Simon (2008): Mental models, consistency and programming aptitude. *ACE '08: Proceedings of the tenth conference on Australasian computing education* Vol. 78.

Cardell-Oliver, R. (2011): How can software metrics help novice programmers? *ACE 2011, Proceedings of the 14th Australasian conference on Computing education*, Perth.

Clear, T. (2008): Thinking issues: assessment in computing education: measuring performance or conformance? *SIGCSE Bulletin* 01/2008; 40:13-15.

Dehnadi., S. (2006): Testing programming aptitude. *Proceedings of the Psychology of Programming Interest Group* 18th Annual Workshop, 22-37.

Fincher, S. (1999): What Are We Doing When We Teach Programming? *29th ASEE/IEEE Frontiers in Education Conference* San Juan, Puerto Rico

Ford, M. and Venema, S. (2010): Assessing the Success of an Introductory Programming Course. *Journal of Information Technology Education* 9:133-145.

Goldstein, H.H., and von Neumann, J. (1947): Planning and coding problems for an electronic computing instrument, part II, vol I. Report prepared for the U.S. Army Ordinance Dept. Reprinted in von Neumann, J. *Collected Works, Vol. V, A.H.* Taub, Ed., Mc-Millan, New York, pp. 80-151.

Gonzalez, G. (2006): A systematic approach to active and cooperative learning in CS1 and its effects on CS2. *SIGCSE 2006*, March 1-5, 2006, Houston, TX, USA.

Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva and Tadeusz Wilusz (2013): A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations *ITiCSE-WGR'13* June 29-July 3, 2013, Canterbury, England, UK.

Ihantola, P., Ahoniemi, T., Karavirta , V. and Seppälä, O (2010): Review of Recent Systems for Automatic Assessment of Programming Assignments. *Koli Calling '10,* October 28-31, 2010, Koli, Finland

Lahtinen, E., Ala-Mutka, K. and Järvinen, H-M. (2005): A study of difficulties of novice programmers. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Monte de Caparica, Portugal, June 27-29, 2005.

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

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. *ITICSE '06 Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*. 118-122

Ma, L., Ferguson, J., Roper, M., and Wood, M. (2007): Investigating the viability of mental models held by novice programmers. *Proceedings of the 38th SIGCSE technical symposium on Computer science education* (SIGCSE '07). ACM, New York, NY, USA, 499-503.

McCartney, R., Boustedt,J., Eckerdal, A., Sanders, K., and Zander, C. (2013): Can First-year Students Program Yet? A Study Revisited *ICER'13,* August 12–14, 2013, San Diego, California, USA.

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), 125-140.

Parsons, D and Haden, P. (2006): Parson's programming puzzles: a fun and effective learning tool for first programming courses, *Proceedings of the 8th Australasian Conference on Computing Education*, p.157-163, January 16-19, 2006, Hobart, Australia.

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. *SIGCSE Bulletin*, 39(4), 2007.

Ramsey, H.R., Atwood, M.E. and Van Doren, J.R. (1983): Flowcharts Versus Program Design Languages: An Experimental Comparison. *Communications of the ACM,* 26(6):445-449.

Sajaniemi, J., Kuittinen, M. and Tikansalo, T. (2008): A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course. *ACM Journal on Educational Resources in Computing,* 7(4).

Scanlan, D. (1987): Data-Structure Students May Prefer to Learn Algorithms Using Graphical Methods. SIGCSE '87 *Proceedings of the eighteenth SIGCSE technical symposium on Computer science education.* pp. 302-307.

Schmuller, J. *Sam's Teach Yourself UML in 24 Hours, Complete Starter Kit* (3rd Edition) Sams Publishing, 2004.

Sheard, J. (2012): Exams in computer programming: What do they examine and how complex are they? In *Proceedings of the 23rd Annual Conference of the Australasian Association for Engineering Education*.

Shneiderman, B., Mayer, R., McKay, D. and Heller, P. (1977): Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM,* 20(6):373-381.

Shneiderman, B. (1982): Control Flow and Data Structure Documentation: Two Experiments. *Communications of the ACM*, 25(1):55-63.

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

Thomas, L., Ratcliffe, M., Woodbury, J., Jarman, E. (2002): Learning styles and performance in the introductory programming sequence . *SIGCSE '02 Proceedings of the 33rd SIGCSE technical symposium on Computer science education.*

Vamplew , P., and Dermoudy, J. (2005): An anti-plagiarism editor for software development courses *ACE '05 Proceedings of the 7th Australasian conference on Computing education* 42:83-90.

Whalley , J., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K. and Prasad, C. (2006): An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies *ACE '06 Proceedings of the 8th Australasian Conference on Computing Education.* 52:243-252.

Wing, J. (2006): Computational Thinking *Communications of the ACM*, 49(3):33-35.

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

Repository of Wisdom : Automated Support for Composing Programming Exams

Keith Foster, Daryl D'Souza, Margaret Hamilton, James Harland
School of Computer Science and Information Technology
RMIT University
Melbourne, Australia

{keith.foster, daryl.dsouza, margaret.hamilton, james.harland}@rmit.edu.au

Abstract

This paper presents a macro-level view around exam composition. While previous work known as BABELnot (Lister et. al. 2012 [1]) developed a micro-level classification scheme to consistently categorise individual exam questions, this paper's contribution uses a more holistic intent towards collective exam composition to build on the past research from the BABELnot project. Specifically, it addresses a higher order, cognitive layer of complexity on top of the exam classification work derived from the BABELnot project to categorise foundation level programming exam questions. In preparation for this, we analysed use cases for a programming questions database for the composition of exams and selected two for further analysis and implementation. A database designed for use by both educators and researchers exists, called "The repository of Wisdom" (RoW), however, it needs further enhancements to support the goals of this paper. The RoW was designed and implemented as part of previous work (Hamilton, D'Souza, Harland, Rosalina 2014 [3]) that classified questions in programming exams. The retrieval of these questions can be by various attributes such as topic, concept, aptitude, content, level of the course and benchmarked results, with interesting and innovative retrieval options related to ranked queries. The selection process can also be influenced by difficulty scores, ratings and comments given by the instructors who submitted the questions or others who may have trialled them. We would like the repository to be further evaluated in the "real world" by computer scientists and, in particular, academics assessing novice programming ability or designing entry level exams. We have built ontologies and mechanisms for storing and retrieving exam questions and also discuss these in this paper.

Keywords: Novice programming; Computer Science Education; Computer programming exam question retrieval; Mastery, Assessment; Educational measurement; Examination generation; and Use case analysis.

1 Introduction

Programming is a fundamental skill required by most computing or information technology degrees, but the extent and level of competence required can vary greatly from one context to another. Recent research (Sheard 2013 [7]) has shown a lack of model-based exam construction for introductory programming. It was concluded, that "while most of the interviewees had heard of at least one pedagogical model for assessment, only one or two designed their exams with specific reference to such a model".

This discovery is the main motivation for our paper: to contribute a useful model of exam construction in the use case of setting summative programming exams.

Programming is also a skill that can cause significant angst amongst the student body, and is often taught in a cumulative manner across a number of foundational courses. Designing an exam to assess students in such courses is not only a task that requires attention to detail, but also one that must be repeated at least once a semester (and possibly more often), and which needs to take into account any prior access students have to similar exams. This can mean that the process of designing exams is sometimes ad hoc, and one that does not necessarily follow systematic principles, or make it easy to show that the desired learning outcomes of the course are what is actually assessed by the exam.

The BABELnot project (Lister et. al. 2012 [1]) was a multi-institutional project whose main thrust was to identify and develop methods by which programming courses could be compared, one of whose outcomes was a formalism for describing and linking learning outcomes to programming exam questions used in introductory and later courses. Another outcome was an archive of exam questions (often including performance data) expressed in this formalism. This lead to the RoW project, in which these questions formed the basis of a web-accessible database of such questions, (Hamilton, D'Souza, Harland, Rosalina 2014 [3]). This database is accessible to all interested academics, and is intended as a resource that can be used to develop exams for introductory programming courses.

Having a database of potentially useful questions is one thing; using this resource effectively to design and develop exam papers for a particular introductory programming course is another. In particular, it is often necessary to consider some potentially competing attributes, such as reliability, validity, difficulty and distinguishability (Angoff 1971 [4], de Klerk 2014 [5], D'Souza, Hamilton and Harland 2013 [6]).

The ProGoSs system (Gluga 2012 [2]) has like-minded aims to automate parts of the University assessment process, however, it is limited in scope to the design of curricula and matching topics with external (government) requirements.

In this paper we aim to explore the methods by which a reliable and valid exam, with the appropriate level of difficulty and appropriate distinguishability, can be developed using the material in the RoW database and how far this can be usefully automated. To this end, we have conducted some deeper analyses of the needs of the users and more importantly incorporated this analysis into the development of the RoW tool to provide better-automated support.

This paper is structured as follows. First, we explain and define some terms for important concepts and meta-data that we refer to in the following sections. Then we present an orderly flow of analysis and identification of use cases of the system in Section 3. User goals are presented in Section 4 and mechanisms to support these goals are presented in Section 5. Finally, with a framework for RoW usage developed we highlight how we intend to implement this in RoW (Section 6) some further work that needs to be done (Section 7) and draw some conclusions (Section 8).

2 Definitions

Here we define the ontologies, concepts and measurements used throughout this paper. In our context, "ontology" is simply a structured set of names either ordered in a list or arranged hierarchically giving semantics to meta-data types for examination questions.

The ontologies presented are *skill elements* - which include *topics*, *concepts* and *aptitudes* - and difficulty *Levels*. The elements divide and classify the notion of skill into atomic components (that a student possesses and/or a question requires) and the levels give a standard approximation of difficulty. These aid in the selection of appropriate questions for an exam.

The concept of *mastery discrimination* is about how well questions distinguish between a student's programming ability and *Partitioning* is how we implement that. These aid in achieving a successfully discriminating exam. The *mastery scale* is our standard linear measurement for both a student's ability and a question's requirements to solve, which consists of *mastery levels* from 0 to 100. Lastly, *Discrimination validity* is a standard linear measurement of the validity of a question's discrimination correctness by a human expert, which is used to help the relevance of query results.

2.1 Skill elements

A skill element is an abstraction of three classifications: *topics*, *concepts* and *aptitudes*, each of which has its own context for relevance. These are defined below:

Topics refer to domain specific topics. For example, The Java switch statement, Model-view-controller, Regular expressions, C++ multiple inheritance.

Concepts refer to generic concepts applicable to many programming topics. For example, Nested loops, Recursion, Memory Pointers, Object references, Class Inheritance.

Aptitudes refer to fundamental brain skills that are considered critical for the ability to programming. For example, Boolean logic, Imperative processes, Mathematical functions, Pattern matching, Analysing written English and structuring / re-phrasing logically.

2.2 Mastery levels

A *mastery level* is simply a value from 0 to 100 attributed to a question that represents the relative amount of ability required to answer the question successfully. This level is intended to correlate closely with exam assessment scores out of 100. So, a student who passes a question at level 50 would be expected to get a score of at least 50 in the overall exam. Similarly students who achieve full marks for a question at level 80 would be expected to get at least 80 for an overall exam score.

2.3 Difficulty levels

Difficulty is a classification of questions to indicate the kind of partitioning the question achieves. That is, a question's ability to discriminate between students of different mastery levels. Different questions partition the set of students at different points on the *mastery scale*. Our proposed ordered list of *difficulty levels* is as follows:

2.3.1 At-risk

This means that if a student fails such a question it indicates they will likely fail overall.

2.3.2 Basic

If a student fails such a question it means they have an error in their understanding that indicates they will likely fail many other questions.

2.3.3 Mainstream

Failing these questions indicate a gap in knowledge, understanding or application that is the part of the core competency of introductory programming. They indicate nothing about the likelihood of passing other types of questions.

2.3.4 Advanced

Passing an advanced question indicates they will likely pass all previous types. Failing indicates they will likely fail challenging questions.

2.3.5 Challenging

These questions serve to discriminate the most able students from the majority.

Difficulty levels are not related to mastery levels, however, we can consider that they will discriminate at some imprecise points on the *mastery scale*:

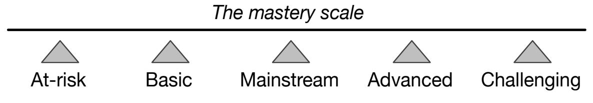


Figure 1: Example Set of Difficulties plotted on the mastery scale.

A secondary use of *difficulty levels* is to aid the exam composer in defining a *partitioning* for each question, which is explained in the following subsection.

2.4 Mastery discrimination and partitioning

Mastery discrimination refers to the ability of a question or an entire exam to distinguish between students of different *mastery levels*. If we test a cohort of students with five questions, each with a different one of the *difficulty levels* above, then we can divide the cohort of students into six subsets based on combined results of all five question as follows:

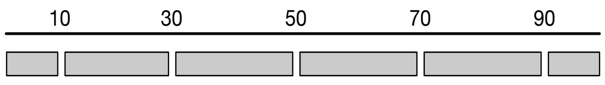


Figure 2: A Five Point Example of Partitioning

Each subset of the cohort contains students that fall into the same range of *mastery levels*. This is assuming, of course, that a student who fails an At-Risk question also fails all higher level questions and the same for Basic and so on. Obviously there will be exceptions to this but statistically this will usually be the case if the questions and their *partitioning* are "good" (i.e. work well for most students).

This process of dividing a cohort into subsets of students based on the student's *mastery level* we call *partitioning* and the subsets we call *partitions*. There is still an ordering of the partitions maintained. A *partitioning point* is a single value in the *mastery level* range that separates two adjacent *partitions*. For example, 10, 30, 50, 70 and 90 (from figure 3).

A **single** question can partition up to a maximum of the number of marks allocated (using one partition point for each mark) plus one.

However, typically and practically, only one partition point is useful (when treating the question as pass/fail for example). So typically, you'll get only two partitions from a single question, even though it may have many more marks allocated to it.

An **exam**, however, can combine the multiple partition points from all of the questions to effectively create a huge partitioning with scores of partition points. This can then partition a cohort into scores of partitions.

In any given exam, the number of partitions is the sum of all the partition points of all questions.

2.5 Discrimination validity

Validity refers to the human element to validate that a given question is in fact a good discriminator based on expertise and past experience. This means an "expert" in teaching programming is needed to review questions and assign a validity score. The Validity score can be used to rank questions from a search result. Ranking is covered in more detail in section 5.2.

3 Use cases

We identify two use cases for composing programming exams with the support of our tool and present each one in the following subsections. In analysing the use cases we draw out a number of goals, some of which are shared between use cases. There were many other use cases that we considered. However, in the interests of focus, we reduced them to these two. Some other use cases are presented in section 8 on further work.

3.1 Composing summative assessment exams

An exam composer is a teaching staff member responsible for writing new exams. In our context, they use the RoW to source questions for a new exam and the metadata incorporated in RoW to guide them in selecting the best or desired questions to include in their exam. We have identified a few goals that an exam composer would have in mind when composing an exam. They include the following.

(Goal 1) Full coverage of the *topics*.

(Goal 2) Full coverage of the *concepts*.

(Goal 3) Full coverage of the *aptitudes*.

Another goal is to have "good discrimination" in an exam meaning the aim is to have the results of a cohort of students to be more distributed across the "mastery scale".

Note that to enable reasoning about student abilities in a standard way, we reuse the hypothetical concept of "mastery level" used in (Gluga 2012 [2]) and defined it as a range from 0 to 100 to be analogous with an assessment result.

For example, a poorly balanced exam may have "clusters" of students with close scores and large gaps between clusters on the result scale. This is undesirable as assessors aim to discriminate between students at many points on the result scale.

We are not claiming that good exams should only display a normal distribution of scores; rather this is an interesting metric that highlights gaps and clusters in the distribution that may indicate a problem with a candidate exam. It would then be the composer's decision whether and how to address these anomalies.

We have identified two different methods by which a composer could improve an exam in this regard:

(Goal 4) Balancing the types of questions in an exam by difficulty.

(Goal 5) Evenly distributing the marks of an exam across all *mastery levels*.

The first is coarse-grained and the second fine-grained. Both are goals for exam composition and are explained further in section 4. The last goal (above) can be satisfied by selecting a number of questions each requiring varying *mastery levels* to pass, such as, including a question to identify the high achievers in the exam, or including a question that bare passing students should be able to do well in. We can expand this last goal to include an additional goal:

(Goal 6) Identifying good *mastery-partitioning* questions

This last goal is really a means to and end, the end being distributing an exam's marks across *mastery levels* using the identified partitioning questions.

3.2 Composing university entrance exams

An entrance exam is one where an educational institution asks prospective students to sit a formal exam prior to making an offer for them to be enrolled into the program. These entrance exams have two main aims. Firstly, to rank the students by their abilities and thereby enable the school to make offers to the top students. Secondly, to partition the students based on aptitude and thereby enable the school to advise the student which program would suit them best (or indeed, if no program suits their aptitudes, they may be advised to seek a different field of study, or undertake some preliminary course of study).

Note that we are not positing nor refuting that the "Geek Gene" (Lister 2010 [8]) determines aptitudes, rather, the measurement of a prospective student's current aptitudes in combination with the University program's aptitude preferences provides relevant information with which to influence the decision. It may well be the opposite case of the "Geek Gene" theory where a University has a program designed to build programming related aptitudes through targeted intellectual exercises and decides to deliberately make an offer to a student who is currently lacking in an important aptitude but who shows great potential on other measures. In both cases, the aptitude assessments are very useful.

In order to generate such an exam there are a few goals we think an exam composer would have in mind:

(Goal 6) Identifying good *mastery-partitioning* questions (for *aptitudes*).

(Goal 4) Balancing the types of questions in an exam by difficulty.

(Goal 3) Full coverage of the *aptitudes*.

Note, we do not include goal G5 - evenly distributing the marks of an exam across all mastery levels - because we think that people don't have a **degree** of mastery in an aptitude, rather, we think the intention of the definition of aptitudes is that people either have an aptitude or they don't.

Concepts and topics are not included in these goals because this is what the students learn, not what is relevant to entrance. A high school student will not have been exposed to many of the concepts usually contained in an introductory programming course at a tertiary education level. Therefore, a specific set of secondary education mathematical and problem-solving *aptitudes,* designed to assess programming *aptitudes* would be more appropriate in this use case.

The set of *aptitudes* for programming should be specific and well known. Identifying a standard set of *aptitudes* for this purpose is another project in itself for further work (Section 7). In the following section we will explain each goal identified in this section (G1 through G6) in more detail.

4 Goals for exam composition

In this section, we briefly explain all the goals required by the use cases identified in the previous section. We explain the expected benefits to the exam composer of each goal in order to highlight what the proposed support mechanisms (in the following Section 5) will need to achieve.

4.1 Full coverage of assessable skill elements

A creator of formative assessments desires an exam that fully covers a set of *topics* and *concepts* that she has decided needs to be assessed. Similarly, a creator of University entrance assessments desires an exam that fully covers a set of *aptitudes* that she has decided needs to be assessed.

The goal is for the set of candidate exam questions to approach 100% coverage of all the specified *skill elements* to be assessed. Effectively there are three goals, one for each sub classification of *skill element*:

(Goal 1) Full coverage of the specified *topics.*

(Goal 2) Full coverage of the specified *concepts.*

(Goal 3) Full coverage of the specified *aptitudes.*

4.2 Optimising mastery discrimination in an exam

In our analysis we identified two approaches to optimising the effectiveness of an exam to discriminate between students with different mastery levels. The first is coarse-grained and uses the difficulty levels defined in section 2. The second is fine-grained and uses *partitionings*, also defined in section 2. Both approaches aim to spread the marks allocated to questions across either difficulty or mastery levels, according to what the exam composer specifies.

(Goal 4) Balancing marks across ***difficulty levels*** **…**

When a composer is using the coarse-grained *difficulty balancing* method to compose a new exam, they desire their exam to contain questions at many different *difficulty levels* and different amounts of marks at each *difficulty level.* To this end, we propose to capture this desire in what we call a "*difficulty signature*". This signature simply defines the proportion (out of 100%) of each *difficulty level* desired. For example: 10% at-risk, 20% basic, 40% mainstream, 20% advanced and 10% challenging. The composer should define the desired *difficulty signature* first and then the system can calculate how closely a candidate exam approximates the desired *difficulty signature*. It is important to note that the percentages above would be weighted on the **marks allocated** to a question, and not on the number of questions.

(Goal 5) *Distributing marks across* **mastery levels …**

When a composer is using the fine-grained *partitioning point distribution* method to compose a new exam they aim to have the graph of marks allocated to questions by *partitioning point* to approach some specified pattern (or graph shape).

Normal distribution is an example of a graph shape, however, a normal distribution is not appropriate for exam discrimination; rather, an even distribution (horizontal line) starting at 30 on the mastery scale may be more appropriate. In a "normal" world, the normal distribution would show itself when plotting the number of students attaining a given score, not the marks requiring that *mastery level*. Therefore, the composer needs to specify the desired graph shape for *mastery level* distribution before applying the metric to a candidate exam.

4.3 (Goal 6) Identifying mastery-partitioning questions

In order to create a candidate exam the exam composer needs to select candidate questions from the database. The RoW tool provides a search mechanism for this and then ranks them according to some pre-defined weighting formula.

This goal is specifically about finding questions that make good mastery discriminators and we use *partitioning* to accomplish this. *Partitioning* refers to the mathematical concept of partitioning a set into two or more subsets. In our context, we want to partition a cohort of students into subsets ordered by *mastery level*. This means every student in a higher subset has a higher *mastery level* than every student in the next lower subset and so on.

Once a set of candidate *partitioning questions* are identified for a particular *skill element* and *partitioning point*, we would like to rank them according to their *discrimination correlation*, popularity, validity and other qualities. The composer uses the ranking to aid in the selection of questions for their exam (higher ranked questions being preferred). We explain how we calculate *partitioning question* ranking in Section 5.2.

We also explain how we can map from a *partition* in any particular question to the standard *mastery level*. A given question may have a score out of five, which results in six partitions. Each partition needs to be mapped to one *mastery level*. How this mapping is achieved is addressed in Section 6.1.

5 Mechanisms to support goals

Having identified the goals that exam composers have, we now describe the mechanics of acheiveing these goals with the RoW tool in mind. We have divided the mecahanisms into four broad categories: searching and ranking questions; then coverage and discrimination analysis of whole exams.

5.1 Searching for candidate questions

In order to create a candidate exam the exam composer needs to search for candidate questions. Most of this functionality is already in the RoW tool allowing her to search by topic, content or level of the course. To this, we add the ability to search by concept and aptitude.

5.2 Ranking of questions

Search results may return many questions – too many to manually review in the selection process – so a ranking of the questions is desirable to reduce the selection time.

One way to rank questions is to compare the ordering of students based on the question's partitioning with the ordering of students based on their overall exam score (we call this *discrimination correlation)*. If the question ordering closely correlates with their overall exam ordering then this could be considered a "good partitioning". If not, then perhaps this question's partitioning is not reliable. However, if the discrimination of a question does closely correlate with the overall exam discrimination, this would not necessarily mean that it is correct, but rather it means that the question's *partitioning point* is consistent with the average scores of the aggregation of questions in past exams.

When ranking *partitioning* questions, we can mix in a number of factors, *discrimination correlation* being just one of them. We have identified four additional factors that could be used in the ranking process:

- **Popularity** - the percentage of "likes" on a question.
- **Validity** - the discrimination validity score assigned by a human expert.
- **Commentary** - a full text analysis of the comments on a question to derive how positive the comments are.
- **Recency** – how recently was the question used in an exam.

In order to rank questions then, an exam composer needs to define a *ranking formula* (or let it default to one) such as this:

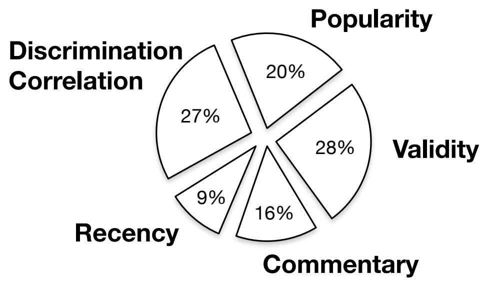


Figure 3: Example *Partitioning Question* Ranking Formula.

Each of these factors will have a relative weight assigned to it for calculating an overall ranking.

5.3 Coverage analysis

Once a candidate exam is established, the exam composer wishes analyse how well it covers all the skill elements desired. Firstly, the composer needs to select a subset of skill elements applicable to this exam (for entrance exams this would likely be aptitudes only). Subsequently the system can then calculate the percentage of the desired elements covered by the set of candidate questions.

The RoW tool's coverage analysis feature should also graph the relative coverage for each element so the composer can visually identify "gaps", "peaks" or "dips" not aligned with the specified coverage graph in order to address them.

5.4 Discrimination analysis

Once a candidate exam is established, the exam composer also wishes to analyse how well it will discriminate between students with varying abilities (measured using mastery levels). As described in Section 4.2, there are two ways to do this: *difficulty balancing* and *partitioning point distribution*. The RoW tool needs a way for composers to pre-define both *difficulty signatures* and *partitioning* graph shapes.

With these created the tool can then calculate the coverage percentage and graph correlation for every question according to any given signature and shape. These calculations are implemented with derived attributes defined in Section 6.2.

6 Tool Support

6.1 Meta-data support

In order to implement the meta-data required by the use cases we propose to expand the “topic” attribute into three classifications (either as three attributes or one attribute of some type with three subtypes): *Aptitude*, *Concept* and *Topic*. For each of these three entities we define the following attributes: *Name* and *Description*. We require each question to have “sets” of aptitudes, concepts and topics associated with them which requires three attributes: Aptitudes required, *Concepts required* and *Topics required*.

As a guide for designing good discriminator questions, it is recommended that questions have very few associated skill elements (ideally one) so the assessment scores and analytics reflect the ability a student has with one specific skill element. Conversely, a question that covers too many skill elements will be ineffective as a discriminator because the assessment scores will be reflecting the student’s aggregated ability with many elements, which cannot be deconstructed back to the elemental level.

For every question, two attributes are defined; *Partitioning* and *Discrimination correlation*. *Partitioning* is a mapping from the marks allocated to a question to the *mastery level*. *Discrimination correlation* is calculated statistically (across all students that sat any exams that included this question). A higher *discrimination correlation* indicates the question performs better as a mastery discriminator.

Partitionings are defined by people adding them to the database and so it requires some further explanation and examples. For each possible assessment, score of the question define the *mastery level* (0 to 100) required to attain that score. For example, a question assessed out of five could have the following partitioning:

Score	Mastery-Level
1	30
2	50
3	60
4	70
5	80

Figure 4: Example Partitioning with six partitions.

A score of zero implies a mastery level less than thirty and a score of one has a mastery level from 30 to less than 50. A different question assessed out of five might have the following partitioning:

Score	Mastery-Level
4	60
5	80

Figure 5: Example Partitioning with three partitions.

A score of zero to three implies a mastery level less than sixty (giving the third partition).

Finally, we need a *Difficulty Level* attribute defined for all questions.

6.2 Analysis support

When creating candidate exams for coverage and discrimination, composers need some information to aid them in the analysis process. Two derived attributes are required on exams: *Skill Coverage* and *Discrimination Correlation.*

Both of these have calculations that involve external data including: desired skill elements, difficulty signature or partitioning graph shape. These three external data items must be defined by the composer prior to running queries involving these attributes.

6.3 Validation support

To validate the correctness of the *discrimination correlation* we need historical data on past question scores by the same students. To support this we define the *Discrimination Validity* attribute for all questions. The value of the *discrimination validity* can be just a number between 0 and 10 which indicates its success rate in discrimination compared to the truth as judged by the expert who enters these validation scores.

6.4 Query support

The two queries described in this section both depend on ranking to be useful. To support this, the RoW tool needs to implement a ranking formula feature that allows composers to alter the weighting of each factor to what they desire. For the purpose of ranking questions for their partitioning qualities, the tool needs to pre-define a derived attribute with a default formula like the one described in Section 5.2. In addition, we need to define the *Discrimination Ranking* attribute for all questions.

Once all the supporting attributes and features are in place, the relevance of useful queries should improve. We have described two useful queries that the tool can perform to realise the use cases identified in Section 3:

6.4.1 Query to fill a gap in coverage

- Filter on Subsets of *topics*, *concepts* or *aptitudes*.
- Group by *Difficulty Level.*
- Order by *Discrimination Ranking.*

The query above is useful to find good discriminator question for an exam. The search can be focused on finding questions for specific skill elements for which the exam currently under construction is lacking. These queries can be iterated and questions collected until full coverage of the desired skill elements is achieved.

6.4.2 Query to fill a gap in difficulty level

- Filter on *Difficulty Level.*
- Group by *Topic* or *concept* or *aptitude.*
- Order by *Discrimination Ranking.*

This query is useful to improve the "balance" of an exam (which in turn aims to get a normal distribution of results for a cohort). The search can be focused on finding question with specific discrimination types for which the exam currently under construction is lacking. These queries can be iterated and questions collected until the desired "discrimination signature" is achieved.

7 Further Work and Reflection

The next phase of work that needs to be performed is creating meta-data about existing exam questions in the repository. We desire to have a significant and useful portion of the questions fully attributed using the above mechanisms and ontologies. To that end, we need experts in teaching introductory programming to create this meta-data and define some ontologies.

Some ontologies might include definitions of standard sets of *aptitudes*, *concepts* and *topics*. In the latter case the set is specific to individual courses so a common set of topics that can be built upon by each educational institution is desirable.

For each question, definitions are required for the *skill elements* and difficulty levels that apply to this question, one partitioning and optionally a discrimination validity for the question. The standard ontologies need to be defined in a focus group consisting of experts. To that end, a forum like an international conference workshop is recommended to complete the job.

Adding values for the above attributes to each question and the subsequent building up of institution/course specific topics can be done incrementally by teachers on their own schedule and in their own places of employment.

Beyond this, we foresee a number of potential directions for the project to progress. We briefly introduce four potential use cases that were not addressed in this paper:

1. Identifying questions that have been used in the past. For example, a computer science educator might find it interesting to compare how their cohort of students is performing in relation to past cohorts of students, or against a cohort from another campus, university or country. This could involve selecting a question or questions for which benchmarked data has already been provided in the RoW, and including this in their own exam, for research purposes.
2. Automatically generating "good" exams given a set of *skill elements* plus a *difficulty signature* and/or a *partitioning distribution graph shape*. The RoW tool could automate the search for the optimal exam that most closely approximates the given *learning attributes*, *difficulty signature* and/or *graph shape*.
3. Analytics on the *discrimination correlation* of individual questions in relation to overall exam scores in order to iteratively refine the question in order to progressively achieve better and better correlation.
4. Collecting data on practical assignments and formative assessments in addition to summative exams for the purpose of analytics. For example, predicting summative results from formative results or at least discovering unknown relationships between them.

In this paper, the mechanisms for improving the mastery discrimination of exams rests on the assumption that questions in an exam are "targeted" (i.e. that require one or only a few skill elements). The reason being that fine-grained allocation of marks to a few specific skill elements greatly aids the accuracy of the partitioning process. If, however, exams contain large questions and marks are allocated broadly across a large number of skill elements then this process breaks down.

To mitigate this flaw in the future, we suggest composers break down large exam questions and allocate smaller amounts of marks to each skill element associated with the question. Commonly, exam markers will have a marking scheme that resembles this already, so the next logical step is to organise these finer-grained marks in a marking scheme based on skill elements and capture results in the RoW. In this way, the discrimination processes benefit and opens the potential for more detailed analytics on individual questions that can improve discrimination at the question level as well.

8 Conclusions

In this paper, we have presented two use case scenarios for developing examination papers based on questions available in the repository of wisdom. We have discussed how mechanisms, ontologies and metrics can be defined for the exam composer to select questions for an exam that satisfy the goals of these different use cases. The *mastery levels* presented in this paper can be mapped to Bloom's taxonomy, or whatever mastery scales the composer requires giving their exam papers the required educational measurement.

It is hoped that the analytical processes presented will result in the dissemination of better questions and exams that discriminate well. As more data is collected about how students have fared using the various questions in the repository, it will become possible to analyse more deeply the relationships between aptitudes, concepts and topics and gain further insights into detecting and overcoming obstacles in learning programming.

9 References

[1] 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., Toward a shared understanding of competency in programming: An invitation to the BABELnot project. In proceedings of the 14th Australasian Computing Education Conference (ACE2012), Melbourne, Australia, 2012.

[2] Gluga, R., Kay, J., Lister, R., "ProGoSs: Mastering the Curriculum", Proceedings of the Australian

Conference on Science and Mathematics Education. (ACSME2012). Sydney, Australia, September 2012 in Australian Conference on Science and Mathematics Education (ACSME2012), ed M. Sharma and A. Yeung, UniServe Science, Sydney, Australia, pp.92-98 http://ojs-prod.library.usyd.edu.au/index.php/IISME/article/view/5914 last accessed: 13.12.13.

[3] Hamilton, M., D'Souza,D.,Harland, H., Rosalina,E., Repository of Wisdom: A database for storing and retrieving classified and benchmarked exam questions for introductory programming courses, Proceedings of the 23rd Australasian Software Engineering Conference (ASWEC), Sydney, April, 2014.

[4] Angoff, W., Scales, norms, and equivalent scores, in R.L. Thomdike (Ed.), Educational measurement (2nd ed., pp. 508-600). Washington, DC: American Council on Education, 1971.

[5] 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.intestcom.org /Publications /ORTA.php, last accessed 1/09/2014.

[6] D'Souza,D., Hamilton, M., and Harland,J., A Comparative Analysis of Results on Programming Exams, Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013) 117-126, Adelaide, January,2013

[7] Sheard, J.,Simon, Carbone, A., D'Souza, D., Hamilton, H., Assessment of Programming: Pedagogical foundations of exams, (2013).

[8] Lister, R., Computing Education Research, ACM Inroads, Vol 1, No.3,2010,http://203.144.248.23/ACM.FT/1840000/1835434/p16-lister.pdf, last accessed 1/09/2011

How (not) to write an introductory programming exam

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Daryl D'Souza
RMIT University
Australia
daryl.dsouza@rmit.edu.au

Mike Lopez
Christchurch Polytechnic Inst of Tech
New Zealand
mike.lopez@cpit.ac.nz

Andrew Luxton-Reilly
University of Auckland
New Zealand
a.luxton-riley@auckland.ac.nz

Iwan Handoyo Putro
Monash University
Australia
iwan.putro@monash.edu

Phil Robbins
Auckland University of Technology
New Zealand
phil.robbins@aut.ac.nz

Donna Teague
Queensland University of Technology
Australia
d.teague@qut.edu.au

Jacqueline Whalley
Auckland University of Technology
New Zealand
jacqueline.whalley@aut.ac.nz

Abstract

The computing education literature shows some recent interest in summative assessment in introductory programming, with papers analysing final examinations and other papers proposing small sets of examination questions that might be used in multiple institutions as part of a benchmarking exercise. This paper reports on a project to expand the set of questions suitable for use in benchmarking exercises, and at the same time to identify guidelines for writing good examination questions for introductory programming courses – and, by implication, practices to avoid when writing questions. The paper presents a set of ten questions deemed suitable for use in the exams of multiple courses, and invites readers to use the questions in their own exams. It also presents the guidelines that emerged from the study, in the hope that they will be helpful to computing educators writing exams for their own courses.

Keywords: introductory programming, CS1, assessment, benchmarking, examination.

1 Introduction

McCracken et al (2001) appeared to discover that many of the students who pass programming courses cannot actually program. The BRACElet project (Whalley et al 2006) explored this issue in great depth and effectively confirmed the problem. Addressing the question of how students might be able to pass programming courses without being able to program, Traynor et al (2006) provided some insight with this excerpt from a student interview: "Most of the questions are looking for the same thing, and you usually get the marks for making the answer *look* correct. Like if it's a searching problem, you put down a loop, and you have an array and an *if* statement. That usually gets you the marks . . . Not all of them, but definitely a pass."

One response to this issue is to analyse the final exams in programming courses, to try to establish how they align with the skills and knowledge that students are expected to acquire. Simon et al (2010) analysed data structures exams in this light, and Petersen et al (2011) and Sheard et al (2013) looked at introductory programming exams.

In an early stage of the current project, 11 common questions were included in the introductory programming exams of six institutions in Australia and New Zealand (Sheard et al 2014). We concluded that four of the questions were suitable for benchmarking purposes, and invited other academics to use these questions in their own exams and compare their students' performance with the published results.

Benchmarking is not an attempt to impose uniformity on courses and assessments across the sector. Rather, it is a way of permitting comparisons: does university A, which has a high reputation and a correspondingly high entry requirement, produce better student outcomes than university B, which accepts the students who are not admitted to the other universities?

Such questions cannot be reasonably asked until there is a meaningful way of answering them. This is what we believe to be the purpose of benchmarking. If interested participants at different institutions can include a reasonable set of common questions in their final examinations, they can compare the results of their students with a published benchmark and form their own conclusions as to the quality of their courses in the context of the student cohorts that they attract.

In reducing an original set of 76 questions to the final 11 (Sheard et al 2014), we noted a number of reasons why participants did not consider questions suitable for use across multiple institutions:

- *Question is too easy.*
- *Question is too large.*
- *Topic is too advanced or not usually covered in a typical introductory programming course.*

- *Student may not be familiar with the style of question.*
- *Style of question is not suitable for an exam situation, e.g. is it reasonable to ask students to identify syntax errors?*
- *Wording of the question is unclear or ambiguous.*
- *Question is idiosyncratic, e.g. referring to the coding style guide of a particular course.*
- *Question involves tricky code, which may obfuscate its purpose.*

In the current phase of the project we set out to further explore these and other reasons, while at the same time expanding the set of questions that can be used for benchmarking. We thus addressed the following questions:

- Can we identify some principles of good question design that others can apply when writing their own questions?
- Can we identify some aspects of poor question design that others can try to avoid when writing their own questions?
- Can we identify examination questions that a group of instructors would all be willing to use in their introductory programming exams?

2 Research approach

The 11 questions from the previous phase of the project were supplemented by a further 20 candidate examination questions, sourced from the literature (principally from publications of the BABELnot project (Lister et al 2012)) and from questions that had been used in exams at the lead authors' institutions. Two additional versions of one question were added, so that the same basic question could be considered in three distinct forms.

The two lead authors conducted a workshop in conjunction with ACE 2014, for academics with current or recent involvement in assessing students in an introductory programming course. The remaining seven authors joined the project by attending the workshop.

The bulk of the workshop consisted of discussion of the 33 questions. For each question, participants rated the likelihood that they would use it in an introductory programming exam, on a scale from 1 (would definitely not use it) to 5 (would definitely use it). At the same time they were asked to give reasons for their choices. Members were at liberty to change their ratings during or after the discussion of each question.

Discussion was lively on many questions, and most members did not complete the rating exercise in the course of the meeting. Members therefore completed the exercise individually in their own time, and submitted their full set of ratings and reasons to the project leaders for analysis.

Analysis began by considering the simple average rating of each question, resulting in a ranking of the 33 questions. This was then supplemented by a qualitative analysis of the members' reasons for their ranking decisions, which resulted in some re-ordering of the list. Finally, questions were selected from the high end of the ranked list, but with consideration to question types and subject matter, so that we did not end up with a substantial number of similar questions.

3 Issues for consideration

In this section we list and discuss issues that arose as we discussed the questions, both at the workshop and in the subsequent data presented for analysis. The issues are in no particular order, and are grouped where possible.

3.1 Question preambles and complexity

Sheard et al (2013) propose a number of measures of question complexity, some of which they suggest should be avoided, while others might be considered a necessary part of what is being tested. One of the measures to be minimised is linguistic complexity, the complexity of the language in which the question is expressed. The essence of the message is that if a question can be expressed more simply, it should be. Among other considerations, this is likely to assist students with a weak grasp of English.

Linguistic complexity is typically encountered in the preamble to a question, the part that sets the scene for what the students are actually being asked to do. Consider Q4, one of the 11 questions from Sheard et al (2014).

Q4. A dependent child can be very loosely defined as a person under 18 years of age who does not earn $10,000 or more a year. An expression that would define a dependent child is

(a) `age < 18 && salary < 10000`
(b) `age < 18 || salary < 10000`
(c) `age <= 18 && salary <= 10000`
(d) `age <= 18 || salary <= 10000`

This question might appear to be expressed in reasonably clear and simple terms. However, one participant questioned the use of the phrase 'very loosely': what did this signify, and might it confuse students into believing that the subsequent definition was not the one to be implemented? In response to this question, the preamble was rephrased to begin "If a dependent child is defined as…". Another participant then queried the use of the word "If", preferring the question to start "A dependent child is defined as…". This wording was rejected on the basis that it appears to be stating a factual definition of dependent children, whereas the intent was simply to provide a definition that could be used for the purposes of this particular question.

There was broader agreement with regard to other questions. For example, the participants all agreed that Q12 would be easier to grasp if the four initialisations were simply presented as the first line of the code, rather than appearing after it with a message telling students to assume that they took place before it.

Q12. This question refers to the following code, where the variables *p*, *q*, *r*, and *s* all have integer values:

```
if (p < q) {
  if (q > 4) {
    s = 5;
  } else {
    s = 6;
  }
}
```

Assume that, before the above code is executed, the values in the four variables are:

```
int p=1; int q=2; int r=3; int s=4;
```

What would be the value in variable *s* after the code is executed?

Q1. It is an odd fact that the more people there are in a group, the less pizza each of them will eat. Using the following code, how many pizzas would you expect 10 people to eat?

```
if people < 5:
  pizzas = people
elif people < 10:
  pizzas = 3 * people / 4
elif people < 15:
  pizzas = 2 * people / 3
else:
  pizzas = people / 2
```

Considerations of linguistic complexity lead to the issue of contextualising questions. Some examiners like to set their questions in some sort of real-world scenario, while others prefer to limit the question to explicit instructions as to what is required of the students. Consider Q1: one participant said of this question that "the first sentence is distracting and not relevant to what the code is asking about"; others expressed similar concerns. One said "if *people* should be initialised to 10, say so explicitly". There appear to be two schools of thought in this regard. One suggests that students should be given instructions solely about what is required, with no superfluous information; the other, that reading and understanding superfluous information is a necessary aspect of problem-solving, and can be legitimately included in programming questions. The participants in this study did not reach consensus on this question.

A related consideration is the explicitness of instruction. Another question mentioned in its preamble that the elements of an array were initialised. One participant wanted students to be told what the initial values were, although this was not relevant to what was subsequently being asked.

Another form of question complexity identified by Sheard et al (2013) is called 'external domain reference'. They noted that some questions refer to subject matter that might not be known to students in an introductory programming course, and they distinguished between cases where such knowledge is integral to the question and cases where it is incidental and can be overlooked. Q19 falls into the latter category, which Sheard et al call medium-level external domain reference. One participant remarked that the "Question requires some real-world knowledge about what payments and balances mean, which may make it difficult for some students". Others presumably felt that the question could be answered even by students lacking that knowledge.

Q19. What is the purpose or outcome of the following piece of code?

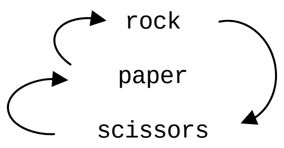
```
for (int i=0; i<payment.Length; i++)
{
  balance = balance + payment[i];
}
```

(a) to add a payment to a balance
(b) to count the payments
(c) to add all payments except the last to the balance
(d) to add all payments to the balance

3.2 Diagrams and examples

In some questions, where it seems that a certain level of complexity is inescapable, diagrams and/or examples can be provided to help students understand the question. Q9 illustrates the point. However, any use of diagrams should be highly contingent on what notation has been used during the course. If students have seen similar diagrams used to explain variable assignment, this diagram would be acceptable; but the final exam is not the place to introduce a graphical notation that the students have not previously encountered.

Q9. There are three integer variables, *rock*, *paper* and *scissors*, which have been initialised. Write code to swap the values in these variables around so that *rock* is given *paper*'s original value, *paper* is given *scissors*'s original value, and *scissors* is given *rock*'s original value. The following diagram illustrates the result of the swaps:



Some participants noted in passing that they were not comfortable with the use of the word 'swap' to indicate movements among more than two items.

When examples are used instead of diagrams or in addition to diagrams, there is a concern that some students will take them as definitive. In Q24, for example, some students might assume that the array will have exactly four elements, and so might write four *if* statements rather than a single *if* statement within an appropriate loop; others might even assume that the code will only be given the array {0, 2, 1, 3}. One participant expressed concern about another question that described an array of unspecified length but gave as an example an array of length 11. But an example is necessarily a particular instance of a generalisation, so it would rarely be possible to provide an example that retains complete generality.

Q24. Suppose you had an array of integers called *mirrors*. Write code that would print out every element of that array that had the same value as its index position. For example, given the array {0, 2, 1, 3}, the code would print the values 0 and 3.

3.3 Material covered in course

It is generally understood that an exam for an early-level course will not test concepts that were not covered in the course. This impacts on our study in that different introductory programming courses do not all cover the same material, even when they are taught using the same language. Questions that are reasonable in the context of one particular course might not be so reasonable in a range of courses at different institutions.

One example of this is the concept of integer division (as in Q1), which one participant describes as "a peculiarity of Java operators being overloaded rather than a core programming concept". It might be reasonable to test the students' knowledge of integer division in a course in which this concept was explicitly taught, but caution should be applied in deciding whether to incorporate the knowledge into questions in other courses.

In addressing our goal of finding a set of questions that can be used in multiple courses using different languages, we quickly decided that input and output must be regarded as off limits. One obvious reason for this is that different programming languages have very different ways of dealing with input and output. A less obvious reason is that different teaching approaches place different emphasis on input and output. For example, an objects-first approach using Java within the BlueJ environment (bluej.org) need not address I/O at all, as the approach focuses on method calls and their results. Similarly, the media computation approach of Guzdial and Ericson (2013) focuses on the input and output of image and sound files, and touches only briefly on keyboard input, many weeks into the course. A code-tracing question with output statements would therefore be better replaced with an output-less version that asks what values certain variables will have when the code has executed.

Terminology will often differ between courses. Q24, in section 3.2, refers to the 'index position' of an array element. In some courses this might simply be called the index, while in others it might be the position. When adopting questions from other courses, great care must be taken to use the terminology that has been used in the target course.

A further consideration is the preparation that students have undergone during the semester. Some of the questions for our study were provided by a participant who gradually prepares the students for such questions with a series of graded exercises throughout the semester. It seems reasonable to expect that this participant's students would perform better on these questions than students who had not been offered the same preparation.

Finally, consideration should be given to any high-level programming tasks that might be provided by the language being studied, and that might have been covered in the course. Simple array-processing tasks that might be tested in an exam include sorting the elements of an array, reversing the order of elements in an array, and finding the average of the elements in an array of numbers. These tasks become somewhat trivial in a language with inbuilt *sort*, *reverse*, and *average* methods. Even if students have not been taught these features, some might have come across them, and might short-circuit the intention of the question by using them in their answers.

3.4 Variable names (and comments) in code

When code is provided as part of an exam question, the author has three options with regard to the variable names: to make them meaningful, neutral, or 'anti-meaningful' (explained below).

Most programming educators impress on their students the importance of using meaningful variable names, and most apply this practice in their own programming (although many seem not to accept that *temp* and *flag* are sadly lacking in meaning). However, meaningful names can lead students to understand code without having to study the code itself. In Q∞ – which was not part of our study – a student with poorly developed code-reading skills would probably be able to deduce the answer just by reading the variable names.

For examination purposes, therefore, some instructors choose to make the names – or at least those names that might give away the answer – neutral. They might leave *person* and *height* there, to tell students that this is a list or array of people's heights, but replace *totalHeight* and *avgHeight* with, say, *value1* and *value2*.

Q∞. What is the purpose or outcome of the following piece of code?

```
totalHeight = 0
for person in range(0, len(height)):
  totalHeight = totalHeight + height[person]
if totalHeight <> 0:
  avgHeight = totalHeight / len(height)
else:
  avgHeight = 0
```

A number of the code-tracing and code-explaining questions in our study included such neutral names. In one question, the code compares two arrays, returning the last index at which the element in the first array is less than the corresponding element in the second. In a similar question, the code counts the number of times the corresponding elements in the arrays are not equal. Several participants expressed concern that the arrays were called *number1* and *number2*, one suggesting that "it would be better with variable names that provided more meaningful context, for example, arrays of coffee consumed in the morning and the afternoon, and counting the number of days when there are unequal numbers of coffee consumed."

On this same point, consider Q12, in section 3.1. One participant wrote of this question "The responses to the question might be different if the variable names were less abstract and had more context. As academics we often abstract away the variable identifiers as being irrelevant to the question, but then ask students to write code that *does* use meaningful variable names, so our assessment is not well aligned with our expectations of practice. I would use this question with meaningful names." Complying with this expressed need for context might then raise another problem: this particular piece of code might have been written with no real-world context in mind. The variables might simply be numbers, not representing any particular quantities. Should the instructor nevertheless contrive some plausible context? Or is it in fact acceptable to ask students to reason about the code itself, without the additional information provided by meaningful variable names?

Instructors who do use neutral names should consider one further issue: are the different names in the code clear and distinct? During the presentation of a paper at ICER 2013 (Ahadi & Lister 2013) the presenter displayed a code-explaining question and asked why so many students answered it wrongly, and one member of the audience murmured "because they're dyslexic?" The code in the question used two variables, *p* and *q*, which are indeed readily confused by people with certain learning difficulties. The same applies to *b* and *d*. Similarly, the commonly used variable *i* is readily mistaken for the digit 1, which can have a serious impact on a student's understanding of a statement such as *count = count + i*. Instructors who are accustomed to reading and understanding code should take care to ensure that it is not open to misreadings of this sort.

As an aside, most instructors also urge their students to imbue their code with explanatory comments. The code provided for code-tracing and code-explaining questions tends to have few or no comments, and certainly does not have comments explaining what the code does. Because the code is therefore not of the standard we expect of our students, does this mean that we cannot ask our students to read and explain it?

Finally, in some of our questions the instructors had used what we might call 'anti-meaningful' names, names that have a meaning, but a meaning that appears unrelated to the purpose of the code, and that might therefore mislead students. Instead of a neutral name such as *number1*, an array might be called *fantasy*. Another example is the name *mirrors* in Q24 (section 3.2). The participant who had contributed this question explained that the code was finding array elements that reflect or mirror their indexes. Nevertheless, other participants found the reference a little obscure, suggesting for example that the name *mirrors* might confuse students into thinking about mirror-images of variables, whatever that might mean. In general, it was clear that most of the participants disliked the use of anti-meaningful names.

3.5 Avoidable obfuscation

All computer code has some inherent complexity. However, any task can be coded in different ways that evince different levels of complexity. Is it reasonable to knowingly express the code in a more complex form to test the students' ability to deal with such a form? Q3 provides a simple illustration of this point.

Q3. What will be the value assigned to the variable *x* as a result of the following statement?

```
   int x = 10+56 / 5+3 % 12;
```

(a) 13
(b) 11
(c) 24
(d) 10
(e) Generates RunTimeException

The justification for this question was that students had been warned to take care with operator precedence, and that this was a reasonable way to test whether they were doing so. Nevertheless, most participants said that they would use this question only if the spacing were uniform throughout the expression.

Obfuscation can also be unintentional. One example of this is the discontinuity of the code in Q12 (section 3.1); another is the perhaps unthinking use of unnecessary code. In general, participants felt that Q6 tested nothing that would not be tested by a shorter code snippet.

Another question asked students to write a loop to print all the numbers between *p* and *q*, inclusive, that are divisible by *N*. Some code provided to scaffold the question included declarations of *p*, *q*, and *N*, declaration of a scanner, and prompt-input sequences for *p*, *q*, and *N*. The general feeling among participants was that it would be better simply to tell students that the variables had been appropriately initialised, rather than giving them unnecessary input/output code to read.

Q6. What will be printed when the following code is executed?

```
   a = 7
   b = 3
   c = 2
   d = 4
   e = a
   a = b
   b = e
   e = c
   c = d
   d = e
 print a, b, c, d, e
```

Another form of obfuscation, or tricky code, is code that looks very like something the students have been taught to use and recognise, but with a subtle twist. The last three lines of Q5 look like the standard three-statement swap, but are in the wrong order, and give the same value to each variable.

Q5. What values will the variables *a*, *b*, and *c* have after the following code has been executed?

```
   int a = 23;
   int b = 11;
   int c = 61;
   a = b;
   c = a;
   b = c;
```

We tend to value students who can form an overview of a piece of code without examining it in detail, but this question has the potential to lure these students into a wrong answer, giving the advantage to the struggling but systematic student who needs to work through the code in detail. All of the participants said that they would be willing to use this question, although some proposed that the problem could be overcome by explicitly asking students to trace the code. However, it was considered preferable to test students' tracing abilities with code that is not so easily mistaken for a recognised algorithm.

3.6 A mix of difficulties

Analysing 20 introductory programming exams from ten institutions in five countries, Simon et al (2012) rated the difficulty of every question as low, medium, or high. While three of the exams they studied had no questions of high difficulty, over the 20 exams, nearly a quarter of the questions were rated at the high difficulty level. Examiners clearly believe it appropriate to include a mix of easy, medium, and hard questions in an exam.

Nevertheless, there are some questions in our study that the participants deemed too difficult. One of these was Q2, Soloway's rainfall problem (Soloway 1986), in what appears to be close to its original formulation.

Q2. Read in integers that represent daily rainfall, and print out the average daily rainfall; if the input value of rainfall is less than zero, prompt the user for a new rainfall.

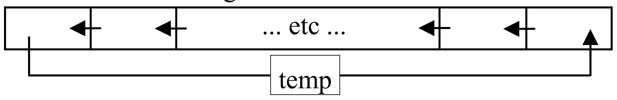
Participants were unanimous that this question was too open, ambiguous, and poorly specified. Some felt that it might be suitable for a practical programming test, but none thought it suitable for a written exam.

Q28, on the other hand, was considered to be difficult but usable. None of the participants expressed concern about the assumption that left represents the lower indexes of the array and right represents the upper indexes – an assumption that is supported by the diagram. The general response was approval (especially when the explicit '5' was removed from the first sentence). The participants liked this question, at the same time acknowledging that this was one of the most difficult questions in the set. That is, they tended to agree with the

Q28. The purpose of the code below is to take an array of numbers (values) containing 5 integers and move all elements of the array one place to the left, with the leftmost element moving to the rightmost position.

```
temp = values[0];
for (int i=0; i<values.Length-1; i++)
    values[i] = values[i+1];
values[values.Length - 1] = temp;
```

For example, if *values* initially has the value [1, 2, 3, 4, 5], then after the code has executed, it would contain [2, 3, 4, 5, 1]. If we were to show the effect of moving all the elements of an array in this way in a diagram, it would look something like this:



Write code that does the opposite of the original block of code above. That is, write code to move all elements of the array values one place to the *right*, with the *rightmost* element being moved to the *leftmost* position.

unspoken notion that an exam should include a mix of easy, medium, and hard questions, and that this question could be one of the last group. Nevertheless, in a subsequent project to use the selected questions in a number of final exams, one instructor decided that the improved version of this question was too difficult and could not be used. It is not clear whether the question was considered too hard even to be one of the exam's more difficult questions, or whether that instructor chooses not to include any difficult questions in exams.

3.7 Form of the question

Most of the exams studied by Sheard et al (2013) included a mix of multiple-choice, short-answer, and code-writing questions, and our question set included examples of all three types.

One issue that does not yet seem to have been addressed in the literature is whether different forms of the same question are equivalent. Our study explicitly addressed this question by including three different forms of the same question, Q29.

Most participants liked the code-writing form of the question, Q29a, with the qualification that some courses might prefer the word 'position' to 'index'.

Q29b, filling in the blanks, was regarded much less favourably. One participant saw it as a trick question that encouraged the students to copy code directly from the first listing to the second, especially as it omits the description of the difference, that is, that the first piece remembers the element while the second should remember the index. Another participant felt that this version was an improvement, removing the potentially confusing wording. A third simply said that students would be horribly confused by this question, while a fourth thought that it might be better as a Parsons problem (Parsons & Haden 2006) – presumably the variant in which multiple options are available for each line of code, as otherwise it could be solved trivially by comparison with the preceding listing.

Q29a. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int best = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < best) best = num[i];
}
answer = best;
```

This code works by remembering, in *best*, the value of the smallest element met so far as it works through the array. Write a piece of code that achieves exactly the same outcome, setting *answer* to the smallest element of *num*, but by remembering *the index of* the smallest element met so far.

Q29b. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int smallest = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < smallest)
  {
    smallest = num[i];
  }
}
answer = smallest;
```

Complete the code in the boxes below so that it also sets *answer* to the smallest element of *num*. Note that the sixth line is different in the two listings.

```
int where = ________;
for (int i=0; i < num.Length; i++)
{
   if num[i] < ________)
   {
     where = i;   // Note difference
   }
}
answer = ________;
```

Q29c. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int best = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < best) best = num[i];
}
answer = best;
```

Which of the following pieces of code does exactly the same thing, that is, sets *answer* to the smallest element of *num*?

```
(a) int best = 0;
    for (int i=1; i < num.Length; i++)
    {
      if (num[i] < num[best]) best = i;
    }
    answer = num[best];

(b) int best = 0;
    for (int i=1; i < num.Length; i++)
    {
      if (num[i] < num[best]) best = num[i];
    }
    answer = num[best];

(c) int best = 0;
    for (int i=1; i < num.Length; i++)
    {
      if (num[i] < num[best]) best = i;
    }
    answer = best;

(d) int best = num[0];
    for (int i=1; i < num.Length; i++)
    {
      if (num[i] < num[best]) best = i;
    }
    answer = num[best];
```

The multiple-choice version, Q29c, was seen by one participant as the best of the options. On the other hand, three believed that it would be too easy to find the answer by strategic guessing or reverse engineering as opposed to reading and understanding the four different pieces of code. It remains an open question whether the strategic guessing or reverse engineering would require students to reason in a similar way as they would if reading and understanding the code pieces, in which case there might not be a problem.

In addition to asking whether participants would use each version of this question in their exams, we asked whether they thought that the three versions were the same, and why.

Nobody thought that they were the same. One participant thought they were equivalent, "essentially but not exactly" the same, and some noted that they were testing the same thing in different ways. Others, however, felt the versions to be quite different as they test different skills: code writing, scaffolded code writing, and code tracing. Most participants thought the multiple-choice version to be the easiest, but one thought that the pure-code writing version was easiest, and two favoured the scaffolded code-writing version.

3.8 Multiple-choice questions

Multiple-choice questions have been the subject of much discussion in the literature, essentially addressing the question of whether they are a legitimate form of assessment. There are guides to writing good MCQs (Hansen 1997, Isaacs 1994), a number of papers proposing how MCQs can be validly used in computing assessment (Lister 2005, Roberts 2006, Woodford & Bancroft 2005), but at least one survey showing that many instructors remain highly suspicious of this question form (Shuhidan et al 2010).

Some participants in our study echoed this suspicion. Of the 33 questions in the study, 11 were presented in the multiple-choice form, and all but three of those drew suggestions that the answers would be too easy to guess, requirements to add further distractors, or both. Some participants who normally use MCQs in their exams expressed no such concerns, but this form of question is clearly still worrying to many instructors.

3.9 Code-explaining questions

A number of the questions in this study ask students to explain the purpose or outcome of a given piece of code.

Q14. Consider the following block of code, where variables *a*, *b*, and *c* each store integer values:

```
if (a > b) {
   if (b > c) {
      Console.WriteLine(c);
   } else {
      Console.WriteLine(b);
   }
} else if (a > c) {
   Console.WriteLine(c);
} else {
   Console.WriteLine(a);
}
```

In one sentence, describe the purpose of the above code (i.e. the if/else if/else block). Do NOT give a line-by-line description of what the code does. Instead, tell us the purpose of the code.

Q19 in section 3.1 and the hypothetical Q∞ in section 3.4 are examples; Q14 is another.

Code-explaining questions were brought into wide use by the BRACElet project (Whalley et al 2006), to test the notion that perhaps students should be able to read code before they can be expected to write code. That project consistently found that introductory programming students had great difficulty deducing the purpose of small pieces of code (Sheard et al 2008, Teague & Lister 2014), even if the questions were presented in multiple-choice form (Simon & Snowdon 2011).

The greatest concern expressed by participants about these questions is their use of non-meaningful variable names. However, as discussed in section 3.4, it would be difficult to provide meaningful variable names without giving away the purpose of the code. Therefore it would seem that neutral variable names might be an unavoidable cost associated with using questions of this type.

With code-explaining questions, as with other questions, it is important to avoid obfuscation. The point can be illustrated with Q14. A knowledgeable programmer might respond that the code prints the smallest value of the variables *a*, *b*, and *c*. Others, however, might wonder how to describe what will happen if two or three of the variables are equal. Would that notion of 'smallest' then strictly apply, and if not, how should they describe which of the equal variables would have its value printed? It is unlikely that these questions were considered by the question's author, yet they have the potential to seriously confuse some students.

Is there, then, any point in setting code-explaining questions? Many appear to think so, and the participants in this study certainly expressed general approval of some of the code-explaining questions provided.

One point that was clearly made by the BRACElet project is that students are less likely to do well on code-explaining questions if they are not familiar with this question type. A final examination is seldom the best place to introduce students to a type of question they have not seen before. Instructors deciding to introduce code-explaining questions to their exams should certainly give students ample prior practice with this type of question.

4 Results: ten questions for broad use

On a scale from 1 (would definitely not use) to 5 (would definitely use), the 33 questions were accorded average ratings ranging from 2.9 (Q2, discussed in section 3.6) to 4.9 (Q5, discussed in section 3.5). Fourteen of the questions, nearly half of them, rated at 4 or above, and only five rated below 3.5.

When participants ranked a question less than 5, their comments sometimes made it clear that they would be happy to use the question with suitable amendments.

We selected ten questions, working from the highest-ranked, so as to produce a mix of question styles and topics. The lowest-ranked question that we selected had an average of 3.6, but was substantially altered (for example, changing it from multiple-choice to short-answer type) to address some of the concerns expressed; the question would therefore have rated more highly if it had been presented in this altered form. All of the other questions chosen had average ratings of 3.9 or higher.

All ten questions are presented in the appendix.

5 Results: how (not) to write an introductory programming exam

The ratings given to the various questions in our study, and the discussion on whether the participants would use each question, lead to a set of guidelines that can be used when writing an exam. The guidelines can be used as a set of positive recommendations, or used in their converse forms as a set of practices to be avoided. Some of these guidelines are already well known, but we believe that there is value in presenting them here as a full set.

Keep questions as simple as possible. Unless you are deliberately making a question complex to test your students' skills in gathering requirements and solving problems, simplify question preambles as much as you possibly can. Then check them to see if you can simplify them still further. Finally, have some colleagues check them, to be sure that they interpret them the same way you do. Include questions in a range of difficulty levels, but be sure that the difficulty of a question is germane, deriving from the inherent difficulty of the task to be performed, not from difficulty in understanding what that task might be.

Consider not contextualising questions. If it is your preference to provide a little real-world (or pretend-world) context for your exam questions, consider whether that context might in fact tend to confuse or mislead students. If it might, consider removing the context so that students will focus on the question you are actually asking.

Use diagrams and examples to help students understand the question. This comes back to the question of what is germane. If it is your goal to see whether students can answer the question, do everything you can, within reason, to ensure that the students understand what the question is. If a diagram or example seems more likely to help students than to further confuse them, provide one. A diagram is far less likely to confuse students if they have seen a number of similar diagrams during the course.

Ensure that students are familiar with the types of question used. It is good to consider adding new question types to an exam, but it might be unfair on the students if the exam is the first place that they see questions of this type. Try to ensure that they have prior exposure to each type of question used in the exam.

When providing code as part of a question, write it as you have taught the students to write. If you have spent a semester trying to teach the students to use good programming style, do not present them with code written in poor style. The exception to this is that neutral variable names should be used if meaningful variable names would give away the answer in a code-explaining or code-tracing question.

Avoid variable names that are easily confused with one another or with other symbols. Consider the ease of confusing *p* and *q*, *b* and *d*, *i* and 1, *l* and 1, *O* and 0; wherever possible, avoid using these single-letter variable names.

Eschew obfuscation. Do not deliberately complicate code. Your exam should determine who can read and understand well-written code – not who can unscramble code that has been written poorly. That skill might be better left for a course on code maintenance.

Include questions of a range of difficulties. Have some easy questions, some moderate questions, and some difficult questions. Easy questions give almost all students a chance to show that they know something about what was taught. Difficult questions, preferably not weighted too heavily, help to distinguish the best students from the rest of the class.

Consider including some multiple-choice questions. It really is possible to write MCQs that test skills other than memory recall, and that distinguish well between the poor students and the good students. They are definitely easier to mark than written-answer questions. And while bright students might be able to deduce the answers by some form of elimination, these are the students who don't need to do so, because they can answer the questions in the way that was intended. Despite the concerns of some of our participants, many students do select wrong answers to MCQs.

Consider including some code-reading questions. Do not assume that your students can read and understand code simply because in a code-writing question they can cobble together an approximation to the answer you were expecting. Be prepared to explicitly test their code comprehension skills.

Include questions of different forms. Be aware of the many different types of question that can be used in an exam, and consider which question type is best suited to each question you intend to ask. Be aware that the same question in different forms is likely to be testing different skills, and choose the form that tests the skills you wish to assess.

6 Conclusions

We set out to answer three questions. Our results show that all three questions can be answered in the affirmative.

Can we identify some principles of good question design that others can apply in writing their own questions? Can we identify some aspects of poor question design that others can try to avoid when writing their own questions? We can and we have. The guidelines in section 5 should be useful to anyone writing an exam, not just in introductory programming but in programming at any level, though of course matters such as question difficulty will need to be adjusted for higher-level courses. Some of the guidelines extend beyond programming, and apply to exam writing in general.

Can we identify examination questions that a group of instructors would all be willing to use in their introductory programming exams? We can and we have. The questions provided in the appendix have been selected on the basis of evaluation by nine academics involved with the assessment of introductory programming courses.

We invite others to include the questions in their own exams, and to either join us in publishing the results, or simply to compare their own students' performance with the benchmark results that we expect to publish. The versions in the appendix are all written in Java, but the project leaders can supply versions of the same questions in C, C#, Visual Basic, Python, and TouchDevelop, and are willing to work on versions for other suitable languages if required. However, we hope it is clear that the questions are not suited to all programming languages, and in particular that they are unlikely to be usable in courses that teach using a functional language and approach.

7 References

Ahadi, A., and Lister, R. (2013). Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant? Ninth International Computing Education Research workshop (ICER 2013), 123-128.

Guzdial, M.J. and Ericson, B. (2013). Introduction to Programming and Computing in Python: a Multimedia Approach, 3rd edition, Pearson Education Inc.

Hansen, J.D. and Dexter, L. (1997). Quality multiple-choice test questions: item-writing guidelines and an analysis of auditing testbanks. Journal of Education for Business 73(2):94-97.

Isaacs, G. (1994). Multiple choice testing. HERDSA Green Guide No 16. Higher Education Research and Development Society of Australasia Inc, Campbelltown, Australia.

Lister, R. (2005). One small step toward a culture of peer review and multi-institutional sharing of educational resources: a multiple choice exam for first semester programming students. Seventh Australasian Computing Education Conference (ACE2005), 155-164.

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. 14th Australasian Computing Education Conference (ACE 2012), 53-60.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study assessment of programming skills of first-year CS students. SIGCSE Bulletin, 33(4):125-140.

Parsons, D. and Haden, P. (2006). Parson's programmimg puzzles: a fun and effective learning tool for first programming courses. Eighth Australasian Computing Education Conference (ACE 2006), 157-163

Petersen, A., Craig, M., and Zingaro, D. (2011). Reviewing CS1 exam question content. 42nd ACM Technical Symposium on Computer Science Education (SIGCSE 2011), Dallas, Texas, USA.

Roberts, Tim (2006). The use of multiple choice tests for formative and summative assessment. Eighth Australasian Computing Education Conference (ACE2006), 175-180.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalleyt, J. (2008). Going SOLO to assess novice programmers. 13th Conference on Innovation and Technology on Computer Science Education (ITiCSE 2008), 209-213.

Sheard, J., Simon, Carbone, A., Chinn, D., Clear, T., Corney, M., D'Souza, D., Fenwick, J., Harland, J., Laakso, M.-J., and Teague, D. (2013): How difficult are exams? A framework for assessing the complexity of introductory programming exams. 15th Australasian Computing Education Conference (ACE 2013), 145-154.

Sheard, J., Simon, Dermoudy, J., D'Souza, D., Hu, M., and Parson, D. (2014). Benchmarking a set of exam questions for introductory programming. 16th Australasian Computing Education Conference (ACE 2014), 113-121.

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(3):229-259.

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. 14th Australasian Computing Education Conference (ACE 2012), 61-70.

Simon and Snowdon, S. (2011). Explaining program code: giving students the answer helps – but only just. Seventh International Computing Education Research Workshop (ICER 2011), 93-99.

Simon, B., Clancy, M., McCartney, R., Morrison, B., Richards, B., and Sanders, K. (2010). Making sense of data structures exams. Sixth International Computing Education Research workshop (ICER 2010), 97-105.

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

Teague, D. and Lister, R. (2014). Blinded by their Plight: Tracing and the Preoperational Programmer. 25th Psychology of Programming Interest Group Annual Conference (PPIG 2014).

Traynor, D., Bergin, S., and Gibson, J.P. (2006). Automated assessment in CS1. Eighth Australasian Computing Education Conference (ACE 2006), 223-228.

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

Woodford, K. and Bancroft, P (2005). Multiple choice questions not considered harmful. Seventh Australasian Computing Education Conference (ACE2005), 109-115.

Appendix: the ten selected questions (renumbered for subsequent use)

Q1. If a dependent child is a person under 18 years of age who does not earn $10,000 or more a year, which expression would define a dependent child?

(a) `age < 18 && salary < 10000`
(b) `age < 18 || salary < 10000`
(c) `age <= 18 && salary <= 10000`
(d) `age <= 18 || salary <= 10000`

Q2. What are the values of *girls*, *boys*, and *children* after the following code has been executed?

```
int girls = 0;
int boys = 0;
int children = 0;
children = girls + boys;
girls = 15;
boys = 12;
```

(a) 0, 0, 0
(b) 0, 0, 27
(c) 15, 12, 0
(d) 15, 12, 27

Q3. There are three integer variables, *a*, *b* and *c*, which have been initialised. Write code to shift the values in these variables around so that *a* is given *b*'s original value, *b* is given *c*'s original value, and *c* is given *a*'s original value. The following diagram illustrates the direction of the shifts:

$$\begin{tikzcd}
a \arrow[dr] & & b \arrow[ll] \\
& c \arrow[ur] &
\end{tikzcd}$$

Q4. What will be the value of the variable *z* after the following code is executed?

```
int x = 1; int y = 2; int z = 3;
if (x < y) {
    if (y > 4) {
        z = 5;
    } else {
        z = 6;
    }
}
```

Q5. Consider the following block of code, where variables *a*, *b*, *c*, and *answer* each store integer values:

```
if (a > b) {
    if (b > c) {
        answer = c;
    } else {
        answer = b;
    }
} else if (a > c) {
    answer = c;
} else {
    answer = a;
}
```

Which of the following sets of values for *a*, *b*, and *c* will cause *answer* to be assigned the value in variable *b*?

(a) a = 1, b = 2, c = 3
(b) a = 1, b = 3, c = 2
(c) a = 2, b = 1, c = 3
(d) a = 3, b = 2, c = 1

Q6. What will be the value of *result* after the following code statements are executed?

```
int[] nums1 = { 1, -5, 2, 0, 4, 2, -3 };
int[] nums2 = { 1, -5, 2, 4, 4, 2, 7 };
int result = 0;
int j = 0;
while (j < nums1.length)
{
    if (nums1[j] != nums2[j])
    {
        result = result + 1;
    }
    j = j + 1;
}
```

Q7. What is the outcome or likely purpose of the following piece of code?

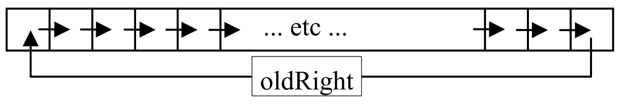
```
int result = 0;
for (int j = 0; j < number.length; j++)
{
    if (number[j] < 0)
    {
        result = result + 1;
    }
}
```

(a) to find the smallest number in the array
(b) to count the negative numbers in the array
(c) to sum the negative numbers in the array
(d) to add 1 to each of the negative numbers in the array
(e) to find the index of the first negative number in the array

Q8. What is the outcome or likely purpose of the following piece of code? Express your answer as a short phrase, like the phrases provided as possible answers in question 7.

```
int result = 0;
for (int count = 1; count <= num; count++)
{
    result = result + count;
}
```

Q9. We can represent an array of integers as a sequence of elements arranged from left to right, with the first element at the left and the last element at the right. Using this representation, a programmer wishes to move all elements of an array one place to the right, with the rightmost element being 'wrapped around' to the leftmost position, as shown in this diagram.



Here is the code that performs that shift for an array referred to by the name *values*:

```
int oldRight = values[values.length - 1];
for (int j = values.length - 1; j > 0; j--)
    values[j] = values[j - 1];
values[0] = oldRight;
```

For example, if *values* initially contains the integers [1, 2, 3, 4, 5], once the code has executed it would contain [5, 1, 2, 3, 4]. Write code that will undo the effect of the above code. That is, write code that will move all the elements of the array one place to the left, with the leftmost element being wrapped around to the rightmost position.

Q10. Write a method that will be given an array of integers and will calculate and return (as a double) the mean (average) of all the integers in the array.

Comparing student performance between traditional and technologically enhanced programming course

Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso,
Rolf Lindén, Einari Kurvinen, Ville Karavirta, Tapio Salakoski
Department of Information Technology &
University of Turku Graduation School (UTUGS)
University of Turku
20014 Turun yliopisto, Finland

{ertaka, temira, milaak, rolind, emarkur, visaka, sala} @utu.fi

Abstract

Educational technology can potentially be used to engage students deeper into learning process, and hence improve the motivation and the learning results. In this paper, we present a study, where an introductory programming course was renewed by using a collaborative learning tool called ViLLE holistically throughout the course. The redesign was done in three main areas: first, half of the lectures were replaced with tutorial sessions, where students completed automatically assessed tasks in collaboration with other students. Second, remaining lectures were accompanied with a group of exercises designed to emphasize the topics introduced. We also collected feedback via short survey after each lecture to find out which topics or issues needed to be addressed again later. Third, the exam was changed into electronic version with automatically assessed programming tasks and questions. When the results of the redesigned course were compared to earlier, traditional instance of the course, we found out, that the pass rates increased significantly, while the average grade remained the same. The results are even more remarkable since the exam in the technologically enhanced course was more complicated than in the earlier instance. Hence, we can conclude that engaging students into active and collaborative learning process has highly positive effect on pass rates, although individual factors cannot be isolated with this many changes in the course design.

Keywords: Programming courses, Introductory programming, Educational technology, Learning environments, Technology adaptation, Student performance

1 Introduction

The educators and researchers in computer science are constantly trying to come up with better means for teaching programming. There have been several studies conducted (see e.g. McCracken et al., 2001, Lahtinen et al., 2005) about the state of programming learning, and in general they seem to come up with worrisome results: the students seem to lack motivation, and the high dropout rates and poor results seem to indicate that there is a lot to do to improve the teaching. Still, limited teacher resources as well as the limited time reserved in curriculum make the course improvement challenging.

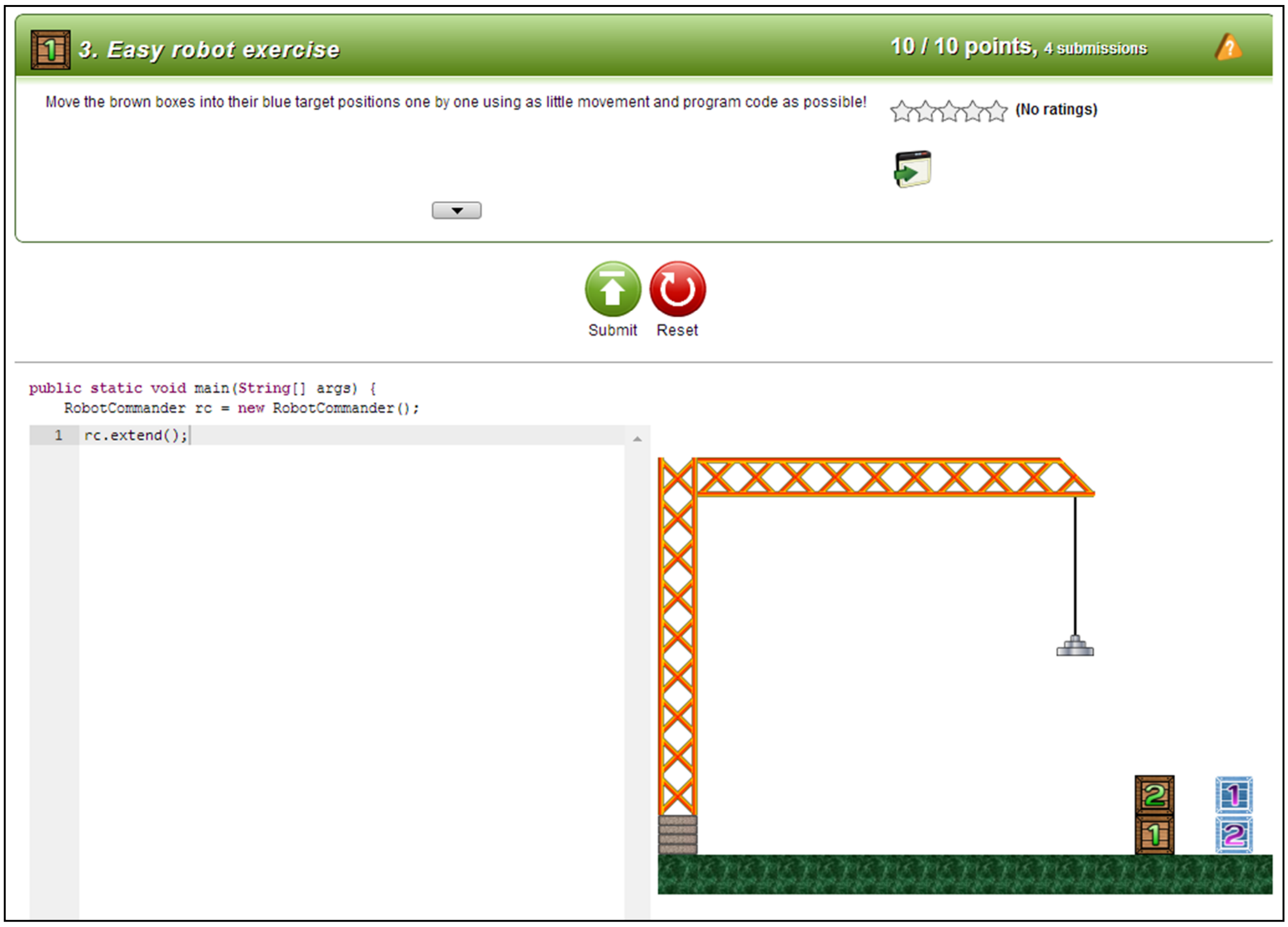
In education, active learning is generally considered as a valid method for engaging students and for improving motivation and results (Freeman et al., 2014). According to constructivist learning theories (see e.g. Papert, 1980, Moons et al., 2013), the knowledge can be constructed by actively participating in the learning process. In programming education this generally means that writing programs and taking other suitable assignments is highly useful in programming educatioin. However, the teachers' workload for assessing several programming assignment in crowded courses can be too heavy.

Educational technology can be used to move the workload away from the course personnel. Automatic assessment and immediate feedback (see e.g. Laakso, 2010) can be effectively used to utilize actively engaging tasks, such as programming assignments. Instead of providing feedback from a few programming assignments in a traditional course, it is possible to offer dozens of automatically assessed tasks by utilizing a novel approach. This means, that the students can be engaged into active learning effectively throughout the course, which presumably means better learning results.

In this paper, we present a redesign of a typical programming course. The change took place between instances of 2011 and 2012. In the redesign the focus was on changing the focus from passive listening into active participation by utilizing educational technology and collaboration. The factors concerning the redesign are discussed as well as the methodology used. Then the performance of two instances of the courses, one right before the redesign and one after, is discussed in the scope of pass rates and course averages.

2 Related Work

As stated in a multinational, multi-institutional study by McCracken et al. (2001), novice programmers lack both motivation and sufficient skills for basic programming after introductory courses. According to Tan et al (2009), the lack of understanding the basic concepts reduces novice programmers' interests for further exploration and self-experimentation in programming. They also state,



Move the brown boxes into their blue target positions one by one using as little movement and program code as possible!

```
public static void main(String[] args) {
    RobotCommander rc = new RobotCommander();
rc.extend();
```

Figure 1: Robot exercise in ViLLE

that novices prefer examples and "drill-practice method", while conventional lectures lead to decreased interest in subject. Lahtinen et al. (2005) surveyed more than 500 students about their difficulties in learning, and found out, that the novice programmers found example programs as most helpful material, and working on exercises most helpful study method for learning to program.

Caspersen and Bennedsen (2007) present a proposition of designing an introductory programming course based on cognitive science and educational psychology. They argue that the cognitive load theory and cognitive skill acquisition play an important part in emphasizing a pattern-based approach to learning. The authors present guidelines in instructional design that they have successfully utilized to redesign the course. Hall et al. (2013) utilized tutorial based learning in the CS course for three weeks, and concluded, that both, tutorials and lectures, should be combined in the course.

Crescenzi and Nocentini (2007) present a two year experiment of utilizing educational technology – namely an algorithm visualization tool – in a programming course. The feedback from students was mainly positive. Still, as reported by Saunders & Kelmming (2003), when technology is integrated into programming course, the students may actually find the module harder, though the performance is improved. According to Rajaravivarma (2005), a games-based approach can be used to emphasize problem solving and logical thinking. In general, engaging students into active learning seems to have a positive effect on motivation and performance.

Utilizing educational technology in a programming course might solve several problems concerning student performance and motivation. There are various learning environments that can be utilized in courses. First, there are the course management systems, such as Moodle (see e.g. Cole et al. 2008) or Blackboard (Bradford et al. 2007). Still, these are traditionally used to manage courses and materials, and in lesser extent to engage students with exercises. Typical examples of exercise-based tools are various visualization tools developed over the recent years. With these tools the users can illustrate the execution of algorithms (see e.g.Grissom et al. 2003, Hundhausen et al. 2007, Malmi et al. 2004) or programs (see e.g. Kannusmäki et al. 2004, Kölling et al. 2003, Oechsle et al. 2002). The visualization is often accompanied with tasks to perform as well.

3 ViLLE

ViLLE is a collaborative learning environment, with focus on exercise-based learning. It supports a variety of exercise types designed for computer science, mathematics, languages and for other topics. All exercises and courses created in ViLLE can be shared with all other teachers registered to system. For CS education, ViLLE supports a variety of programming languages, including for example Java, Python, C++ and C#.

ViLLE supports collaboration in two ways: first, it enables students to work together with one computer, solving the exercises in collaboration. This method utilizes the best practices of pair programming (see e.g. McDowell et al., 2002, Beck & Andres, 2004.), but can be utilized with other types of exercises as well. Second, all resources (courses, exercises and tutorials) created in ViLLE can be shared with other teachers easily. This means, that it can be used for distributing best practices with other educators.

The exercise types found most suitable for the course redesign are

- **Coding exercise:** an exercise where a student is supposed to write a program or a missing part of the program code in given programming language. The solution is tested against model solution provided by the teacher, and the test cases can be randomly parameterized.
- **Robot exercise:** a special version of coding exercise, where a student needs to write a program that controls a robot crane. The goal is to move a number of boxes into their target positions (Figure 1).
- **Visualization exercise:** an exercise where the program code is executed one step at a time, and the execution is visualized with various components – including variable values, object states and call stack. The execution is accompanied with multiple choice questions, open questions and graphical array questions.
- **Simulation exercise:** an exercise where student needs to simulate the state of the program one step at a time by creating variables and objects, changing their values and references and handling the methods in the call stack.
- **Code sorting:** also known as Parson's puzzles (Parsons et al. 2006). A student needs to organize the shuffled program code lines into the correct order according to given task. The solution can be visualized after the sorting, if there are no errors in the program.
- **General sorting:** an exercise where a student needs to sort or connect objects as required. For example, connecting result values with expressions, or value ranges with object types.
- **Quiz:** contains multiple choice questions and open questions.

We have previously researched the usage of ViLLE in various studies with promising results. As shown in Kaila et al. (2009), ViLLE can be used effectively to enhance learning in various different setups and with different methods. The effect achieved on controlled setups was transferred into course-long usage in Kaila et al. (2010) and Kaila et al. (2014), where we demonstrated, that student performance can be significantly improved if ViLLE is integrated holistically into the course.

The complete description of the environment as well as more use cases can be found in the ViLLE system paper (Laakso et al, 2014), and at ViLLE home page (http://ville.cs.utu.fi).

4 Course redesign

Introduction to algorithms and programming is a compulsory programming course for first year CS majors at University of Turku. The course contains fundamental programming concepts – such as variables, conditional statements, repetition, methods and arrays – in Java. In addition to CS majors, several other students from the faculty take the course as mandatory part of their minor studies. For most students, the course is the first actual programming course, though some very basic concepts of programming in Python are covered in an introductory course before that. Course lasts for eight weeks, and 5 ECTS are awarded for passing it. The course methodology was thoroughly redesigned between instances of 2011 and 2012 (from now on C2011 and C2012). In this section, the differences between instances are presented.

4.1 Facilitating active learning with tutorials

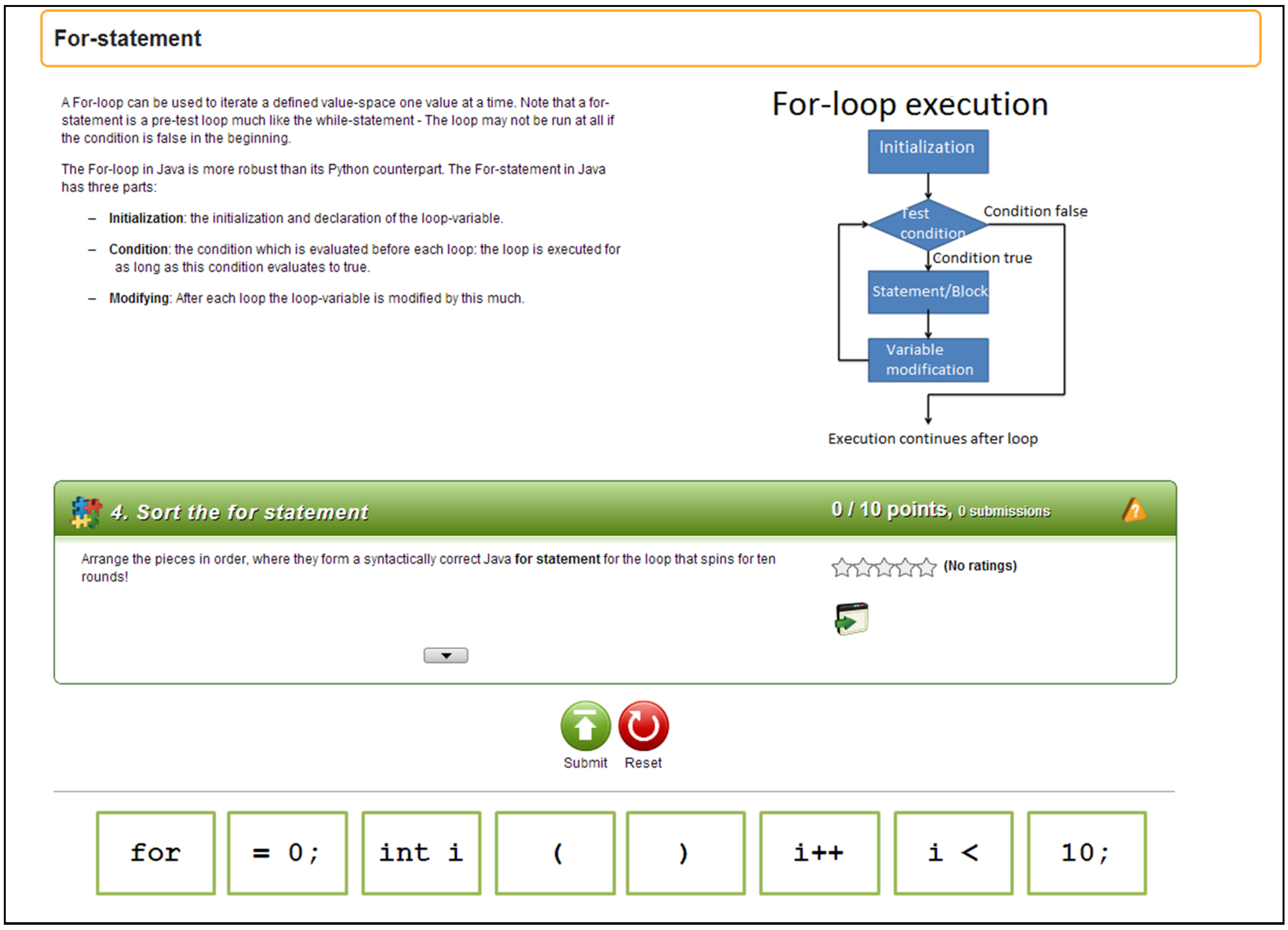
The first, and probably the most important, step was to introduce a concept of more active learning by using tutorials. In the 2011 instance of the course, there were two 2-hour lectures each week. In C2012, one of the lectures each week was replaced with a tutorial-based active learning session. The tutorials were created in ViLLE, and consisted of different types of assignments combined with related learning material such as text, tables and images. Hence, each week consisted of a two-hour lecture about the topic in hand and a two-hour tutorial session, where the topics presented at the lecture were rehearsed. In total, seven tutorials were prepared:

1. Course introduction, advancing from Python to Java
2. Variables, Strings and conditional statements
3. Loops
4. Methods
5. Arrays
6. Using existing classes and modules
7. Summary about all topics

The tutorial sessions were organized in a lecture hall, where students brought their own computers. The tutorials were taken in collaborative mode, where two students worked on the same computer. Both students were awarded points from each solution. The controller – i.e. the student using mouse and keyboard – was switched every fifteen minutes to ensure active participation of both students. Active discussion was encouraged, and at least four members of course personnel were present in each session to assist the students with their possible problems.

Each tutorial consisted of nine to thirteen ViLLE assignments accompanied with learning material, adapted from the lecture slides. Roughly half of the assignments were coding exercises, while the other part consisted of visualization, code sorting, simulation and quizzes. An example of tutorial view is displayed in Figure 2**.**

Each tutorial was open for one week, but the collaborative mode was disabled after the two-hour session. Minimum of 50 % of maximum points as well as participation in at least five of the seven tutorial sessions were made mandatory to pass the course.


For-statement

A For-loop can be used to iterate a defined value-space one value at a time. Note that a for-statement is a pre-test loop much like the while-statement - The loop may not be run at all if the condition is false in the beginning.

The For-loop in Java is more robust than its Python counterpart. The For-statement in Java has three parts:

– **Initialization**: the initialization and declaration of the loop-variable.

– **Condition**: the condition which is evaluated before each loop: the loop is executed for as long as this condition evaluates to true.

– **Modifying**: After each loop the loop-variable is modified by this much.

4. Sort the for statement — 0 / 10 points, 0 submissions

Arrange the pieces in order, where they form a syntactically correct Java **for statement** for the loop that spins for ten rounds!

```
for   = 0;   int i   (   )   i++   i <   10;
```

Figure 2: Tutorial view in ViLLE

4.2 Underlining the importance of lectures with ViLLE exercises and surveys

Around three to four simple ViLLE exercises were prepared to accompany each week's lecture. The exercises consisted of a quiz about the topics covered in lecture, a simple simulation or coding exercise, and a survey. The same three questions were included in each survey:

1. What did you learn from this week's lecture?
2. What things remain unclear after this week's lecture?
3. How would you develop this week's lecture?

The data was analyzed each week before the next lecture, and the results were facilitated instantly: for example, the issues listed as unclear were summarized at the beginning of the next lecture. Also, several small technical problems were fixed based on student feedback.

Each of the exercises were scored with maximum of 5 to 10 points (surveys giving automatically full five points if answered), and the students were required to gain at least 50 % of total maximum points to participate in the final exam. In addition, ViLLE was used to automatically record the student attendances in lectures by using RFID readers in lecture halls and RFID tags given to each student. Though the participation in lectures was not mandatory, some bonus points were awarded if a student participated in all of them.

4.3 Redefining testing with electronic exam

In C2011 the final exam of the course was answered traditionally with pen and paper. Typically the exam consisted of three questions: two programming tasks (done in paper), and a theoretical question, such as an essay. In C2012 the exam was transformed into electronic form by using ViLLE. There are several benefits in using the electronic exam in a programming course:

1. An electronic exam can be automatically assessed, meaning less work for the teacher and quicker access to results for the students.
2. Programming exercises can be done by actually typing, testing and debugging the programs instead of writing them on paper.
3. More heterogeneous exercise types can be used, including for example simulation, visualization and code sorting exercises.
4. Even if manually assessed questions are to be used, they are easier to type and edit with a computer; also, the answers are easier to read and assess compared to those answered in pen and paper.

To make sure that the new instance of the course was comparable – or at least not easier – than the old one, the new electronic version of the exam was created as more challenging. A typical version of the exam in C2012 consists of seven programming tasks – one being a robot task, a quiz measuring theoretical knowledge, and a sorting or simulation exercise. The comparison of exams is displayed in Table 1.

C2011: Exam with pen and paper	C2012: Electronic exam
Manually assessed by teacher and course assistant(s)	Fully automatically assessed
Two programming tasks	Seven programming tasks
One theoretical question	One quiz of 10 MCQ / open questions and one code sorting or simulation exercise
Duration: four hours	Duration: three hours

Table 1: Comparison of exams in C2011 and C2012

The exams in C2012 were evaluated in the same scale than in C2011: minimum of 50 % of points was required to pass – i.e. to get grade 1. After that the subsequent grades of 2…5 were awarded in linear scale. The exam instances were evaluated by four individual researchers and/or teachers not affiliated with this paper, and they all agreed that the new instance is at least as difficult as the earlier instance, and very likely even more challenging.

The electronic exam was organized in one lecture hall and two computer labs at the same time. In the lecture hall the students used their own laptops, while the department computers were utilized in the computer labs. All internet traffic went through a firewall, and the only sites allowed during the exam were ViLLE and Java API. There were practically no technical difficulties during the exam, probably because the students had been familiarized with the setup during the tutorial sessions.

4.4 Other components in the course

Other changes in the course were somewhat minor. For example, C2012 contained the same number of demonstrations than C2011. In demonstrations, the students present their solutions to the programming tasks they are given a week before. In both instances at least 50 % of demonstration score needed to be achieved to attend the final exam. Only technical change in latter instance was that ViLLE was used to record the demonstration points by using aforementioned RFID readers and tags.

Also, the lectures were given in the same traditional form in both instances. However, as there was only half the number of lectures in C2012 – as half of the lecture times were used for tutorials – and the same topics needed to be covered, the lecture content needed to be compacted. Lecture content and slides were modified slightly after C2012 for the following years, based on the student feedback collected via surveys.

5 Course performance

Course performance was studied in one instance (C2011) of the traditional course as well as one instance (C2012) of the redesigned course. The instances are displayed in Table 2.

	C2011	C2012
Course time	October to December, 2011	October to December, 2012
Methodology	Traditional	Renewed
N	210	193

Table 2: Course instance properties

As seen on the table, the number of students starting the course was similar in both instances. However, as is typical for any programming course, not all of the students made it to the exam. The requirements to qualify for the exam are listed in Table 3.

C2011	C2012
50 % of demonstration points	50 % of demonstration points 50 % of tutorial points 50 % of ViLLE exercise points Participation in minimum of 5 tutorial sessions

Table 3: Requirements to qualify for course exam

The number of students who completed the required parts of the course to qualify for the exam and participated in at least one of the exams are displayed in Table 4.

	C2011	C2012
N	210	193
Students participating in exam	149	167
% of all students in exam	70.95 %	86.53 %

Table 4: Percentage of students qualified to final exam

Notably there were more students qualified to take the final exam in the latter instance though there were more requirements to qualify.

In both courses, there were three possibilities to take an exam. A student could take the exam more than once, regardless of whether (s)he had passed the earlier exams. Combined final results in both instances are displayed in Table 5.

Grade	C2011	C2011 proportion	C2012	C2012 proportion
5	45	30 %	70	42 %
4	21	14 %	19	11 %
3	20	13 %	23	14 %
2	12	8 %	19	11 %
1	14	9 %	25	15 %
Fail	37	25 %	11	7 %
Total	149	100 %	167	100 %

Table 5: Grade distribution in course instances

The distribution is visualized in Figure 3.

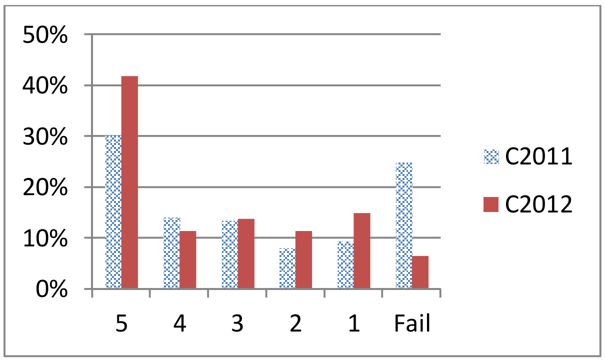


Figure 3: Grade distribution in course instances visualized

As seen in the table and the figure, the most significant difference in distribution among instances seems to be at the highest and the lowest grades. This is also the explanation for the grade average remaining the same; it is likely, that the active learning methods helped a lot of "worst" students to pass the course in the new instance.

The combined results for both instances are displayed at Table 6**.**

	C2011	C2012
Total N	210	193
Qualify to take exam	70.95 %	86.53 %
% passed exam (of qualified)	75.17 %	93.41 %
% passed course	53.33 %	80.82 %
Grade mean (of passed)	3.63	3.57
Grade std. dev.	1.53	1.41

Table 6 Course performance results

As seen on the table, all pass rates in C2012 were significantly higher than in the earlier instance. Still, the grade average remained almost the same between instances.

To confirm the difference, the grade distribution was analysed against a null hypothesis "the distribution of grades is the same across two groups". With significance level of 0.05, we were able to reject the null hypothesis with both, Mann-Whitney U Test (p=0.004) and Kolmogorov-Smirnov test (p=0.011).

6 Discussion

Based on the student performance on the course, it seems that the redesign was quite successful. There was a significant raise in the pass rate as well as in the number of students who qualified to- and passed the final exam, respectively. Curiously, the grade average remained almost the same between the instances. It hence seems that though more students qualified for exam and passed the course, the increase in pass rate was not achieved at the cost of the performance in the final exam. Remarkably, the final exam in C2012 was likely more complex than the one on C2011: instead of two programming assignments, there were now seven. The assignments were at the same difficulty level, in fact, some of the programming tasks from C2011 were used in the exam at C2012.

What reasons may have affected the increased performance? First, the main reason is probably the introduction of active learning methods. As seen before in various studies (see e.g. Laakso, 2010) learning is more efficient when students are actively engaged into the process instead of passively following a lecture. The tutorial sessions seemed to work even better than what we have hoped for: the student feedback collected each week was mainly positive – only concerns being some technical aspects, such as network errors. The students also discussed the topic very actively during the sessions. This seems to be in line with our earlier observations (see Rajala et al. 2009, Rajala et al. 2010): we have previously shown that visualization has a more significant effect on learning when used in collaboration with another student, and that when students engage into using visualizations in collaboration, almost all discussion concerns the topic at hand.

Still, even after the redesign, half of the lectures were kept in the curriculum. The concept behind the redesign was to connect the theory and the practice by offering one lecture and one tutorial session each week. Whether transforming all lectures into active learning sessions would have had similar – or even better – effect remains unknown in the scope of this research. Still, it is definitely a concept worth testing in the future. To underline the significance of certain topics at lectures, a few ViLLE exercises were introduced after each lecture. The quiz about the introduced topics, as well as a simple coding or simulation task, was meant for summarizing the lecture. The survey about the concepts learned and improvement suggestions were also meant for students' self-reflection: it is likely, that analysing and structuring the concepts right after the lecture can have a positive effect on learning.

Automatic assessment was a key factor in course redesign. Without the obvious benefits of automatically assessing programming assignments, the usage of exercises to this extent would have been virtually impossible. Though tutorials were primarily solved in the dedicated tutorial sessions, most of the students needed to complete some of the assignments outside the class room. Automatically assessed programming assignments also provided students a chance to redo tasks later for practice. Also, using ViLLE to try out simple Java programs is easier than starting an IDE or using compiler in command line.

Another important factor in the redesign was immediate feedback provided in ViLLE. When doing the assignments the students got feedback right after clicking the submit button. This also meant that when doing programming tasks at tutorials or weekly exercises, they could compare their results against the model solution results right after submitting, and keep on modifying their program until the results matched. As previously shown in Laakso (2010), automatic assessment and immediate feedback are the key factors when using educational technology effectively. In the earlier instance the only feedback students received from their programs was during the demonstrations. A student got to present his/her solution probably once or twice during the whole course; when compared to more than hundred automatically assessed tasks done in the latter instance, with unlimited number of submissions, this difference can probably be seen as the most significant reason for the performance differences.

Immediate feedback was not provided in the course exam. Still, the students could see the compiler and runtime errors to bring the programming process closer to actual programming, testing and debugging. The students also had access to Java API. Moreover, the students got a subtle visual feedback if the answer was 100 percent correct: the background colour of the coding area changed to light green. Actually, this feature was left originally in exam mode as a mistake, and as such the students were not notified of it beforehand. Still, at least some of the students reported it as a nice feature in the final exam, since it helped them to confirm that their solution was correct. All programming assignments were randomly parameterized, and the test cases always checked for null and empty values and overflows, meaning that regardless of the visual feedback, the students could not test random solutions for full score. Moreover, as only automatic assessment was utilized, the students did not score any points on submissions that could not be compiled.

The student feedback on the novel features was highly positive. According to weekly surveys, the students seemed to value the tutorial based learning over all other forms of teaching. Moreover, a short survey was conducted after the course exam: according to results, the students faced no technical problems, thought that ViLLE as an exam platform was easy to use, and would recommend ViLLE usage to other students. When asked whether they would rather take the exam in paper, only 5 % of the students answered yes.

To conclude, the effect of the redesign seems to be highly positive. Still, there are various factors not considered in the scope of this paper. Most importantly, we can't isolate the effects of individual changes in the new design. Although the change should be observed as holistic, it would be interesting to try to isolate the factors that have the best effect on learning. Also, the student feedback is not comprehensively analysed in this research, as the focus is on performance effects after the redesign. These, to name a few, are definitely factors we will observe closer in the future studies. In the future, we also plan to utilize tutorial-based learning in other CS courses, starting from the introductory course to computer science and algorithms. The method is also going to be tested at other universities, including for example RMIT at Melbourne, Australia.

7 References

Beck, K. & Andres, C. (2004). *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional

Bradford, P., Porciello, M., Balkon, N. & Backus, D. (2006) The Blackboard Learning System: the be all and end all in educational instruction? *Journal of Educational Technology Systems*, 35, 3, 301-314.

Caspersen, M & Bennedsen, J. (2007). Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research* (ICER '07). ACM, New York, NY, USA, 111-122

Cole, J. & Foster, H. (2008). *Using Moodle: Teaching with the Popular Open Source Course Management System (2nd ed.).* Sebastopol: O'Reilly Media Inc.

Crescenzi, P. and Nocentini, C. (2007). Fully integrating algorithm visualization into a cs2 course: a two-year experience. *Proc. of the 12th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Dundee, Scotland, June 25 - 27, 2007). ITiCSE '07. ACM, New York, NY, 296-300.

Freeman, S., Eddy, S., McDonough, M, Smith, M., Okoroafor, N., Jordt, H. & Wenderoth, M. (2014) *Active learning increases student performance in science, engineering, and mathematics.* PNAS 2014*;* published ahead of print May 12, 2014.

Grissom, S., McNally, M. and Naps, T. 2003. Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. In *Proceedings of the ACM Symposium on Software Visualization*, San Diego, California, 87-94.

Hall, S., Fouh, E., Breakiron, D., Elshehaly, M., & Shaffer, C.A. (2013). Evaluating Online Tutorials for Data Structures and Algorithms Courses. In *Proceedings of the 2013 ASEE Annual Conference & Exposition,* Atlanta, GA, June 2013

Hundhausen, C.D. and Brown, J.L. 2007. What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners. Journal of Visual Languages and Computing, 18, 1, 22-47.

Kaila, E., Rajala, T., Laakso, M.-J. & Salakoski, T. (2009). Effects, Experiences and Feedback from Studies of a Program Visualization Tool. *Informatics in Education*, 8, 1, 17-34.

Kaila, E., Rajala, T., Laakso, M.-J. & Salakoski, T. (2010). Long-term Effects of Program Visualization. In *12th Australasian Computing Education Conference* (ACE 2010), January 18- 22, 2010, Brisbane, Australia.

Kaila, E., Rajala, T., Laakso, M.-J., Lindén, R., Kurvinen, E. & Salakoski, T. (2014). Utilizing an Exercise-based Learning Tool Effectively in Computer Science Courses. *Olympiads in Informatics 8.*

Kannusmäki, O., Moreno, A., Myller, N. and Sutinen, E. 2004. What a Novice Wants: Students Using Program Visualization in Distance Programming Course. In *Proceedings of the Third Program Visualization Workshop (PVW'04)*, Warwick, UK

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. 2003. The BlueJ system and its pedagogy. Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, 13, 4.

Laakso, M.J, Kaila, E. & Rajala, T. (2014) ViLLE: designing and adapting a collaborative exercise-based learning environment. Sent to *Computers & Education*

Laakso, M.-J. (2010*). Promoting Programming Learning. Engagement, Automatic Assessment with Immediate Feedback in Visualizations.* TUCS Dissertations no 131.

Lahtinen, E., Ala-Mutka, K. & and Järvinen, H.-M.. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05)*. ACM, New York, NY, USA, 14-18

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O. and Silvasti, P. 2004. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, 3, 2, 267-288.

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, 125-140.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. In Proceedings of the 33rd *SIGCSE technical symposium on Computer science education* (SIGCSE '02). ACM, New York, NY, USA, 38-42.

Moons, J. & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education* 60, 368-384.

Oechsle, R. and Schmitt, T. 2001. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *Revised Lectures on Software Visualization, International Seminar*, May 20-25, 76-190.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY, USA: Basic Books, Inc.

Parsons, D. & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06)*, Denise Tolhurst and Samuel Mann (Eds.), Vol. 52. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157-163

Rajala, T., Kaila, E., Laakso, M.-J. & Salakoski, T. (2009). Effects of Collaboration in Program Visualization. Appeared in the *Technology Enhanced Learning Conference 2009 (TELearn 2009)*, October 6 to 8, 2009, Academia Sinica, Taipei, Taiwan.

Rajala, T., Salakoski, T., Kaila, E. & Laakso, M-J. (2010). How Does Collaboration Affect Algorithm Learning? A Case Study Using TRAKLA2 Algorithm Visualization Tool. In *Proceedings of 2010 International Conference on Education Technology and Computer (ICETC 2010)*, Jun 2010. [A4]

Rajaravivarma, R. (2005) A Games-Based Approach for Teaching the Introductory Programming Course. *Inroads – The SIGCSE Bulletin.* 37, 4, 98-102.

Saunders, G. & Kelmming, F. (2003) Integrating technology into a traditional learning environment. *Active Learning in Higher Education* 4: 74–86.

Tan, P.-H., Ting, C.-Y & Ling, S.-W. (2009). Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception. *International Conference on Computer Technology and Development* (ICCTD '09).

Comparative Study on Programmable Robots as Programming Educational Tools

Shohei Yamazaki[1] **Kazunori Sakamoto**[2] **Kiyoshi Honda**[1]
Hironori Washizaki[1] **Yoshiaki Fukazawa**[1]

[1] Department of Computer Science and Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan
Email: shohei-hamamatsu@moegi.waseda.jp, khonda@ruri.waseda.jp, washizaki@waseda.jp, fukazawa@waseda.jp

[2] Department of Computer Science and Engineering
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
Email: exkazuu@nii.ac.jp

Abstract

Computational Thinking skills are basic and important to manipulate computers. Currently, several systems exist to provide an effective way to learn programming that use computers, smartphones, tablets, or programmable robots. Although studies have reported improved programming skills and motivation to learn programming using an on-screen application or a programmable robot, the benefits of these tools have not been directly compared.

To resolve this issue, especially with regard to motivation to learn programming and impression of programming, we conducted a large-scale comparative experiment involving 236 middle and high school students to evaluate the effects of a game-based educational application and programmable robots on learning programming. We then compared the effects of a game-based educational application with and without programmable robots on learning programming. We found that employing programmable robots on learning programming did not always give an improvement to all students.

Keywords: comparative study, programming education, programming environment, programmable robot, motivation, impression

1 Introduction

Computers have become commonplace. Because of this, Wing has suggested that people should learn Computational Thinking, which she defines as basic skills for manipulating computers (Wing 2006). Thus, we developed educational tools that teach computational thinking.

The motivation to learn and the impression of learning contents are very important not only when developing computational thinking, but learning in general. Several studies have focused on the importance of motivation to learn programming (DeClue 2003, Feldgen & Clua 2004, Kelleher et al. 2007, Jenkins 2001). Feldgen and Clua argued that instructors are critical in motivating students (Feldgen & Clua 2004). Jenkins argued that motivation is the product of expectation and value; thus, students must expect to succeed in learning and value their achievements (Jenkins 2001). These studies demonstrate the importance of providing learners with expectations and the value of being able to program.

Several educational tools have been developed to provide motivation to learn programming (Kölling & Henriksen 2005, Esper et al. 2013, Bezakova et al. 2013). For example, Scratch is a visual and block-based programming learning environment that allows learners to learn programming intuitively (Resnick et al. 2009). Several studies have investigated Scratch (Rizvi et al. 2011, Lewis 2010). Malan and Leitner as well as Maloney et al. have reported the effects of using Scratch as a programming educational environment on learning programming (Malan & Leitner 2007, Maloney et al. 2008). In addition, programmable robots have the potential to facilitate and inspire motivation to learn (Nourbakhsh et al. 2000, Lalonde et al. 2006). In fact, several studies have used robots as educational tools (Kumar & Meeden 1998, Billard et al. 2008). One such robot is LEGO®Mindstorms®. Those learning programming using LEGO Mindstorms create a robot by combining sensors and motors. Barnes reported a study in which Java was taught using Lego Mindstorms as a programming educational tool (Barnes 2002).

Although it is clear that introducing these learning environments and educational tools into learning programming is effective, the following remains unclear. Do these educational tools improve motivation to learn programming? Do these tools improve the impression of programming? How much is the actual improvement using these tools?

In this paper, we evaluate the effects of a game-based educational application and programmable robots on learning programming. We gathered 236 middle and high school students, most of whom were unfamiliar with programming, to participate in our experiment. Then we compared the effects of a game-based educational application with and without programmable robots on the motivation to learn programming and the impression of programming.

The contributions of this paper are:

- We conducted a large-scale comparative experiment where 236 students learned programming.
- We compared the effects of a game-based appli-

cation with and without programmable robots on the motivation to learn programming and the impression of programming using a questionnaire containing six items.

- We investigated the gender differences of the effects of programmable robots furthermore.

The rest of this paper is organized as follows. Section 2 details related works. Section 3 describes the game-based application, while two different programmable robots are described in Section 4. Section 5 details the comparative experiments. The results are evaluated in Section 6. Finally, our conclusion and future work are detailed in Section 7.

2 Related Work

Several studies have examined the effects of programming educational tools and environments on learning motivation. For example, there are several programming educational environments (Kelleher et al. 2007, Long 2007, Kölling & Henriksen 2005, Esper et al. 2013, Bezakova et al. 2013). Additionally, several studies have employed programmable robots as programming learning tools (Nourbakhsh et al. 2000, Lalonde et al. 2006, Fagin et al. 2001, Magnenat et al. 2012). Although they demonstrated the effects of teaching programming concepts to students without programming experience, the influence of game-based applications with and without programmable robots on learning were not compared.

McNally et al. investigated the motivation of two student groups at university (McNally et al. 2006). One group participated in LEGO Mindstorms activities, while the other took a traditional introductory programming course. The difference between our study and McNally et al. is that they discussed the motivation of undergraduates already familiar with programming. Our study investigates not only the motivation but also the impression of programming for middle and high school students, most of whom are unfamiliar with programming.

Scratch, which is aimed at novice programmers, was created by a group at the MIT Media Laboratory in collaboration with a group at UCLA (Resnick et al. 2009). Rizvi et al. investigated the effect of using Scratch to improve the retention and performance of at-risk computer science majors (Rizvi et al. 2011). The difference between these studies is that they targeted undergraduates majoring in computer science and investigated differences between students enrolled in CS0 and CS1, while we investigated the motivation to learn programming and the impression of programming of individuals unfamiliar with programming.

Lewins compared the effects, especially attitude and learning programming concepts, using either Logo or Scratch for sixth grade students learning programming (Lewis 2010). Although the Logo environment seemed to support students' confidence, interest in programming, and understanding of loop constructs, Scratch improved students' understanding of the construct conditions. These studies only treated on-screen applications, whereas our comparative study involves both an on-screen application and a programmable robot.

Previous studies have not compared the effects of game-based educational applications with and without programmable robots on learning to program as long as we investigated. Thus, we conducted such a comparative study with an emphasis on the motivation to learn and the impressions of programming.

3 Game-based Educational Application

We developed an educational tool called ManekkoDance (Sakamoto et al. 2013). ManekkoDance is a programming educational tool that runs as an application on a smartphone or a tablet. There are two reasons why we developed an educational application for a smartphone or a tablet instead of a desktop or laptop computer. First, mobile applications can motivate students (Mahmoud 2008). Second, learning can occur anytime and anywhere using a smartphone or a tablet rather than a computer. ManekkoDance is a game where users move two yellow and orange baby chicks and answer problems by imitating the movements of two white and ocher chickens correctly as models by programming. For example, if the chickens raise their right wings, users have to raise the baby chicks' right wings. ManekkoDance shows whether the user program is correct (see Figure 1).

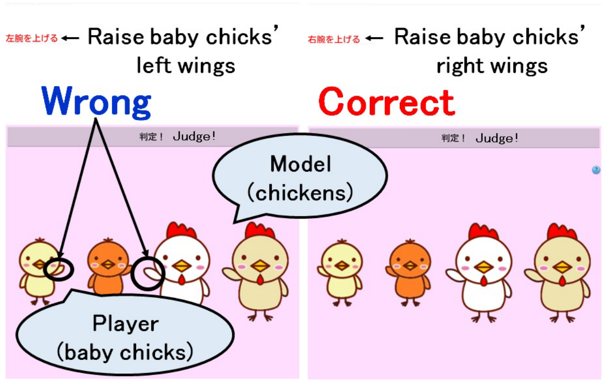

左腕を上げる ← Raise baby chicks' left wings
右腕を上げる ← Raise baby chicks' right wings
Wrong
Correct
判定！ Judge!
判定！ Judge!
Model (chickens)
Player (baby chicks)

Figure 1: Screenshot of ManekkoDance (Left and right show an incorrect and correct program, respectively)

Users can play ManekkoDance, even if users connected programmable robots or did not connect programmable robots. Thus, we adopted ManekkoDance in this experiment.

To understand our experiments, here we briefly describe the features and learning contents of this application.

3.1 User Interface

A previous study reported that a good user interface can motivate learners (Cho et al. 2009). ManekkoDance has appealing interfaces such as the baby chick and chicken characters and icons which move baby chicks. Several students said, "The icons and characters are lovely or cute."

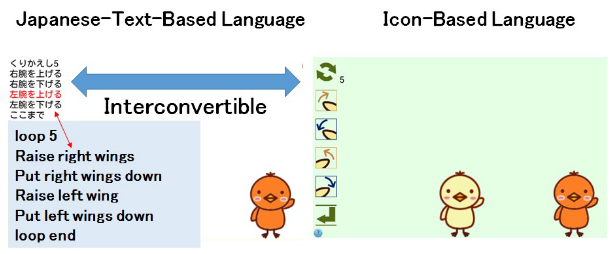
3.1.1 Icon-based Non-verbal Programming Language

Figure 2 shows that ManekkoDance interconverts between a verbal language and icon-based nonverbal programming language, allowing users to more easily write and intuitively understand a program.

Figure 3 shows sixteen icons that correspond to the baby chicks' actions. To play the game, users employ these sixteen icons and natural numbers. Users also have the option to use verbal language.

3.1.2 Characters

To prevent boredom while learning to program, we adopted appealing characters. For example, if the



```
くりかえし5
右腕を上げる
右腕を下げる
左腕を上げる
左腕を下げる
ここまで
```

```
loop 5
Raise right wings
Put right wings down
Raise left wing
Put left wings down
loop end
```

Figure 2: Same program written in a Japanese-text-basaed language (left) and icon based language (right)

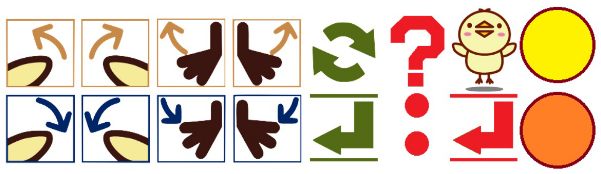


Figure 3: Sixteen icons

written program contains an error, instead of an error screen, the baby chicks fall down. Programming an unnatural motion gives rise to errors. For example, entering a icon to raise the baby chicks' right wings when their wings are already raised causes the baby chicks to fall down.

3.2 Learning Contents

We think that computational thinking is a common concept to various programming languages. We are referring to their idea about computational thinking (Brennan & Resnick 2012).

This game consists of stages so that users can learn gradually. The stages require users to combine the following four concepts in computational thinking. By playing the game, users can learn four concepts in computational thinking that are used in common in many programming languages:

- Sequences
- Concurrency
- Loops
- Conditionals

To view the flow of a sequence, the executed line is sequentially highlighted by a red letter in the execution screen. This allows users to comprehend sequences.

If a user enters plural icons in the same line, the program runs simultaneously. For example, if a user enters two icons in the same line to raise the right and left wings, the baby chicks simultaneously raise both wings. Therefore, users can learn concurrency intelligibly.

Most programs contain a loop function. Thus, in ManekkoDance, users can employ a loop function if they want the chicks to repeat a motion. Figure 4 shows the example program of a loop function in this game.

For example, if a user would like to repeat a chicks' motion, a program is inserted between a loop command, which consists of the starting symbol and a natural number to indicate the number of times to repeat the motion, and a green ending symbol. One

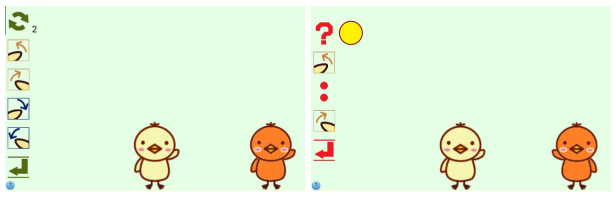


Figure 4: Example programs of loop functions (left) and conditionals (right)

stage requires that a user writes a program so that the baby chicks repeat the motions to raise their left wing, their right wing, put their left wing down, and put their right wing down. This repeated sequences teaches the convenience of the loop function.

Conditionals are important concepts that are used frequently in programming. Users can learn the conditional concept by choosing to move only one of the baby chicks. Figure 4 shows the example program of conditionals in this game. The conditional command consists of the following rules. A user must enter a red question mark, which means "if", and yellow or orange circle which means yellow or orange baby chick in the same line. A red colon means "else". Conditionals end at a red symbol. For example, conditionals make the yellow chick raise its right wing while the orange chick raise its left wing (see Figure 4).

4 Programmable Robots

As mentioned in Section 2, several programming educational tools such as programmable robots have been developed. The processing result of the program written by a learner is not only reflected in the software but also in the robot (e.g., LEGO Mindstorms), which a learner can see and touch. To evaluate the effects between game-based educational applications (on screen) and programmable robots on the ability to learn programming, we conducted a comparative experiment with an emphasis on motivation to learn programming and impression of programming.

By connecting Manekko Dance and two robots, a user can operate the two robots from ManekkoDance. For example, if a user writes a program to move the baby chicks' right wing, the two robots raise their right hands as well (see Figure 5). Because a student may dislike a particular robot, we used two different programmable robots. That is, we avoided things that could decrease motivation to learn or negatively impact impression of programming.

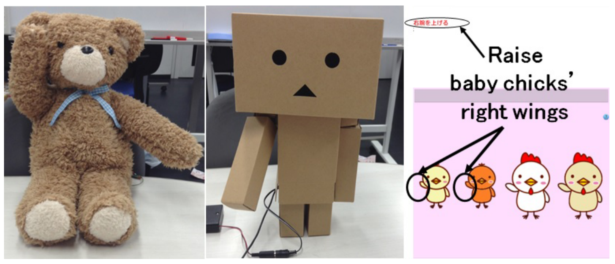


Figure 5: Two Robots interlocked with ManekkoDance (Stuffed Teddy Bear Robot, Cardboard Robot and screenshot of ManekkoDance on left, center and right sides, respectively)

4.1 Stuffed Teddy Bear Robot

We used a Stuffed Teddy Bear Robot (STBR) (Takase et al. 2013) which can move its head and hands as well as roll its head.

STBR has two features: an appealing appearance and a soft texture. This robot is a cuddly teddy bear with fluffy fur. Takase et al. argued that the fluffiness is a factor of loveliness (Takase et al. 2013). Additionally, STBR is so soft that a user can strongly grasp it. Its moving parts consist of fabrics such as cloth, thread, and cotton. The fluffy fur is a factor that makes STBR soft to the touch.

Figure 6 shows the connection of STBR and ManekkoDance, which uses a Wireless Fidelity (Wi-Fi) and a Web application. STBR, a personal computer (PC), and a smartphone or tablet are connected through Wi-Fi. The PC functions as a Web server. The application on the smartphone or tablet sends the signal to move STBR to the PC, which then sends the signal to STBR.

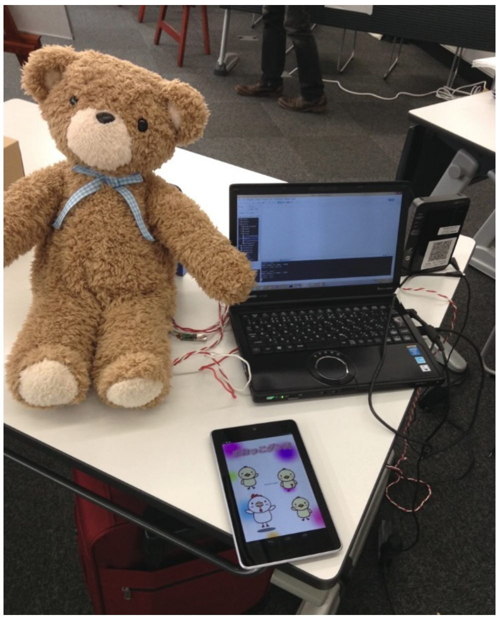


Figure 6: STBR connected with ManekkoDance

4.2 Cardboard Robot

We also used a Cardboard Robot called DANBOARD™, which is a popular character that appearing in Japanese comics. The Cardboard Robot can move its hands differently from STBR. The Cardboard Robot has two main features: a pretty appearance that is not a typical robot and a form that is familiar to users.

Figure 7 shows the connection of Cardboard Robot and ManekkoDance. Moving the servomotor attached to this robot's arms via a pulse wave allows its arms to be raised and lowered. The Cardboard Robot is connected to a smartphone or tablet through the earphone jack.

5 Experiment

We conducted a large-scale comparative experiment involving 236 middle and high school students who were inexperienced programmers attending an open campus event at our university on August 2 and 3,

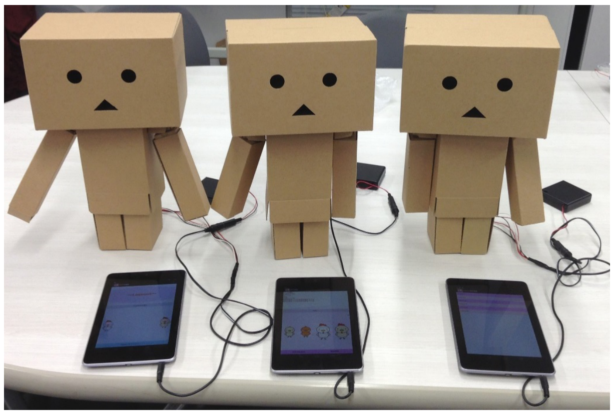


Figure 7: Cardboard Robot connected with ManekkoDance

2014. Open campus is an event in which an individual can participate freely in Japan. We asked students about programming experience by the before questionnaire.

Some students used one STBR connected to ManekkoDance, others used one of the three Cardboard Robots connected to ManekkoDance and the others used ManekkoDance alone as educational tools. To evaluate the effects of a game-based educational application and programmable robots on learning programming, we randomly divided the students into three groups by distributing numbered tickets. Students were divided into three groups according to the numbered tickets (Table 1):

Group A: Each student who learned programming using only ManekkoDance.

Group B: Each student who learned programming using STBR connected to ManekkoDance as a programmable robot.

Group C: Each student who learned programming using a Cardboard Robot connected to ManekkoDance as a programmable robot.

Group	Boys	Girls	Total
A	76	35	111
B	38	23	61
C	41	23	64
B&C	79	46	125
A&B&C	155	81	236

Table 1: Numbers of people participating in this experiment

Each student completed a questionnaire before and after participating in the experiment. For each student, we compared the responses of these two questionnaires and analyzed the effects of a game-based educational application with or without programmable robots on learning from two viewpoints: the motivation to learn programming and the impression of programming.

The experimental procedure was the same for all groups. First, students completed the before questionnaire. Then they learned programming using the tools based on group assignment. Finally they completed a survey after the experiment. The experiment lasted 30 minutes per student. The questionnaire contained six questions. In addition, we classified the

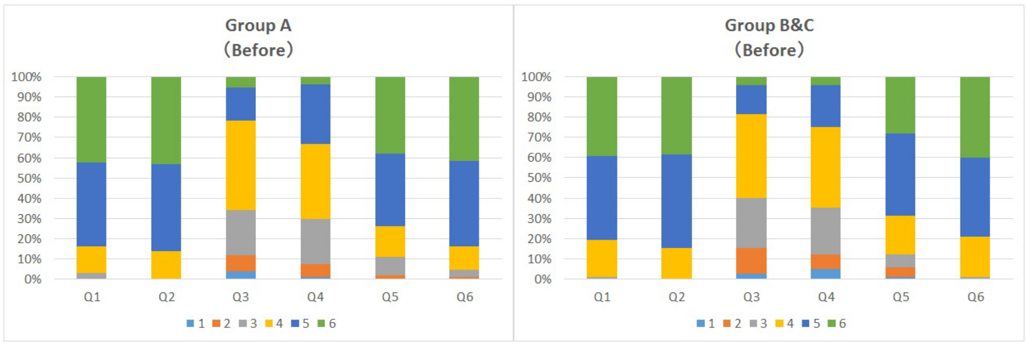


Figure 8: Bar graph of the results of Group A and Groups B&C prior to the experiment. Color scales denote a rating of 1(strongly disagree)6(strongly agree), respectively. Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness)

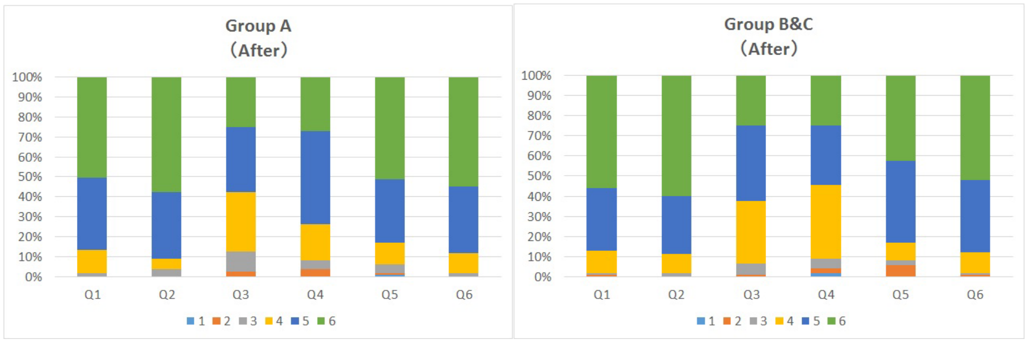


Figure 9: Bar graph of the results of Group A and Groups B&C after the experiment. Color scales denote a rating of 1(strongly disagree)6(strongly agree), respectively. Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness)

motivation to learn and the impression of programming into six question items more finely as follows:

Q1: I want to learn programming. (motivation)

Q2: I feel that programming is fun. (impression)

Q3: I think that I can program. (self-confidence)

Q4: I think that liberal arts students can do programming. (liberal arts)

Q5: I think that being good at programming are related to gender. (gender)

Q6: I think that programming skills are useful. (usefulness)

6 Evaluation

We evaluate the results of our experiment and answer following RQs:

RQ1: Does using a game-based application and a programmable robot result in a difference in motivation and impression of learning programming?

RQ2: Compared to a game-based application, does using a programmable robot increase the rate of positive responses to Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness) in the survey?

6.1 Results

We evaluated the before and after questionnaires to compare the effects of a game-based application with and without programmable robots on the motivation to learn programming and the impression of programming.

	Before		After		After − Before	
	Q1B	Q2B	Q1A	Q2A	Q1A−Q1B	Q2A−Q2B
a_1	4	5	6	6	2	1
a_2	3	4	6	5	3	1
Average					2.5	1

Table 2: Example of the subtraction method

Group	Q1	Q2	Q3	Q4	Q5	Q6
A	0.117	0.153	0.901	0.901	0.261	0.216
B&C	0.216	0.240	1.152	0.880	0.336	0.192
Change Rate (B&C/A)	1.844	1.279	1.567	0.977	1.286	0.888

Table 3: Average of the subtraction results

For the comparison, the responses from Groups B and C were combined and compared to the responses from Group A for the six items described in the previous section (Q1 Q6). All of the students replied to the questionnaires on a six-point scale where a six indicated strongly agree and a one indicated strongly disagree.

Figure 8 shows the ratings prior to the experiment, while Figure 9 shows the ratings after the experiment. The figures employ color scales where aqua, orange, gray, yellow, blue, and green denote a rating of 1 6, respectively.

Because directly comparing the raw data (Figures 8 and 9) did not clearly demonstrate differences between answers regarding motivation and impression of programming, we employed a different analysis approach. For each question, we subtracted the value before from the value after the experiment for each person. Table 2 shows an example using Q1 (Q2) where Q1B (Q2B) and Q1A (Q2A) denote before and after the experiment, respectively, while a_n denotes individual responses. For example, if a_1 answered 4 to Q1 before the experiment and 6 after the experiment, the net value is 2. Then the average difference was determined using all the responses for Group A and Groups B&C.

Table 3 and Figure 10 show the average values of the subtraction method for all six questions. In Figure 10, blue and orange indicate Group A and Groups B&C, respectively.

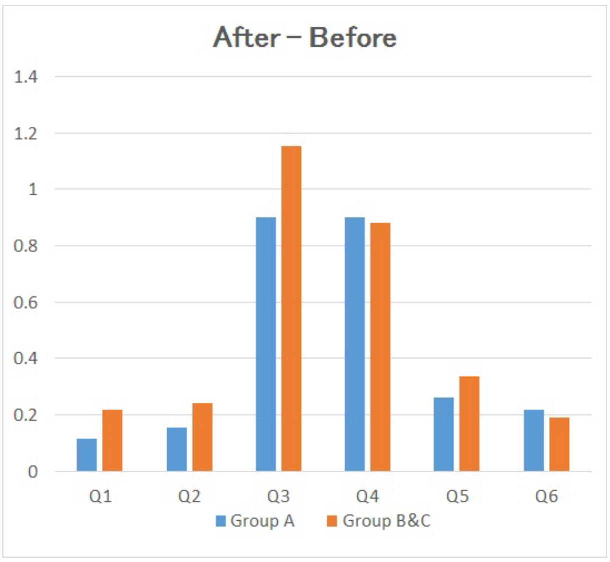


Figure 10: Bar graph of the average of the subtraction value. Blue and orange indicate Group A and Groups B&C, respectively. Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness)

6.2 Discussion

In Table 3 and Figure 10, RQs can be answered.

RQ1: Differences clearly exist between using a game-based application with and without a programmable robot.

RQ2: Q1) Employing programmable robots increases the positive responses to Q1 (motivation) 1.844 times more compared to a game-based application alone. Programmable robots may motivate students to learn programming compared to a game-based application alone.

Q2) Employing programmable robots increases the positive response to Q2 (impression) 1.279 times more compared to a game-based application alone.

Q3) Employing programmable robots increases the positive response to Q3 (self-confidence) 1.567 times more compared to a game-based application alone. Moving programmable robots connected to a game-based application may provide students with self-confidence compared to a game-based application alone.

Q4) Employing programmable robots slightly decreases the positive response to Q4 (liberal arts) (0.977 times) compared to a game-based application alone. Liberal arts is almost changeless when programmable robots are compared to a game-based application alone. We discuss the result about liberal arts later.

Q5) Employing programmable robots increases the positive response to Q5 (gender) 1.286 times more compared to a game-based application alone. We discuss the result about gender later.

Q6) Employing programmable robots decreases the positive response to Q6 (usefulness) (0.888 times) compared to a game-based application alone. Q6 (usefulness) may be ineffective because programmable robots can act only simple things. For example, programmable robots can move only both hands.

Liberal Arts: Andersen et al. reported that fewer liberal art students are interested in programming compared to science students (Andersen et al. 2003). Although the average value with regard to Q4 (liberal arts) decreases when using a programmable robot, most of the students participating in the experiment have not settled on a major. Thus, Q4 (liberal arts) may be ineffective for the participants. Because the students participating in the experiment have not settled on a major, we cannot go into detail about the differences between liberal arts majors.

Group	Gender	Q1	Q2	Q3	Q4	Q5	Q6
A	Boys	0.118	0.197	0.947	0.987	0.184	0.211
B&C	Boys	0.316	0.266	1.076	0.848	0.329	0.228
A	Girls	0.114	0.057	0.800	0.714	0.429	0.229
B&C	Girls	0.043	0.196	1.283	0.835	0.345	0.130
Change Rate (B&C)/A	Boys	2.672	1.347	1.136	0.859	1.787	1.082
	Girls	0.380	3.424	1.603	1.309	0.812	0.571

Table 4: Average subtraction values by gender

Gender: The less number of girl students who, major in computer science has become a problem (Olivieri 2005). Thus, we considered that girl students would not be interested in programming compared to boy students. However, Q5 (gender) in Table 3 and Figure 10 shows that the programmable robots have a positive result on the average change. To investigate the gender difference, we divide the results of the before and after questionnaires by gender. Table 4 and Figure 13, 11 and 12 show the results.

For Q2 (impression of programming) and Q3 (self-confidence) the average change when using a programmable robot increases for both genders. Additionally, for Q2 (impression of programming) and Q3 (self-confidence), it is more effective for girl students to employ programmable robots than for boy students. Especially, for Q2 (impression of programming), while the boys' average change is 1.347, the girl' is 3.424. It is more effective for girl students to employ programmable robots compared to boy students because the girls' average change is 2.54 times of boys'.

For Q1 (motivation), Q5 (gender) and Q6 (usefulness), the boys' responses increase, while the girls' decrease. For Q1, while the boys' average change is 2.672, the girls' is 0.380. It is more ineffective for girl students to employ programmable robots compared to boy students because the boys' average change is 7.031 times of girls'. For Q5 (gender), in Table 3, employing programmable robots increases the positive response to Q5 (gender) 1.286 times more compared to a game-based application alone was obtained. In detail, while the girls' average change was 0.812, the boys' was 1.787. For Q6 (usefulness), while the boys'

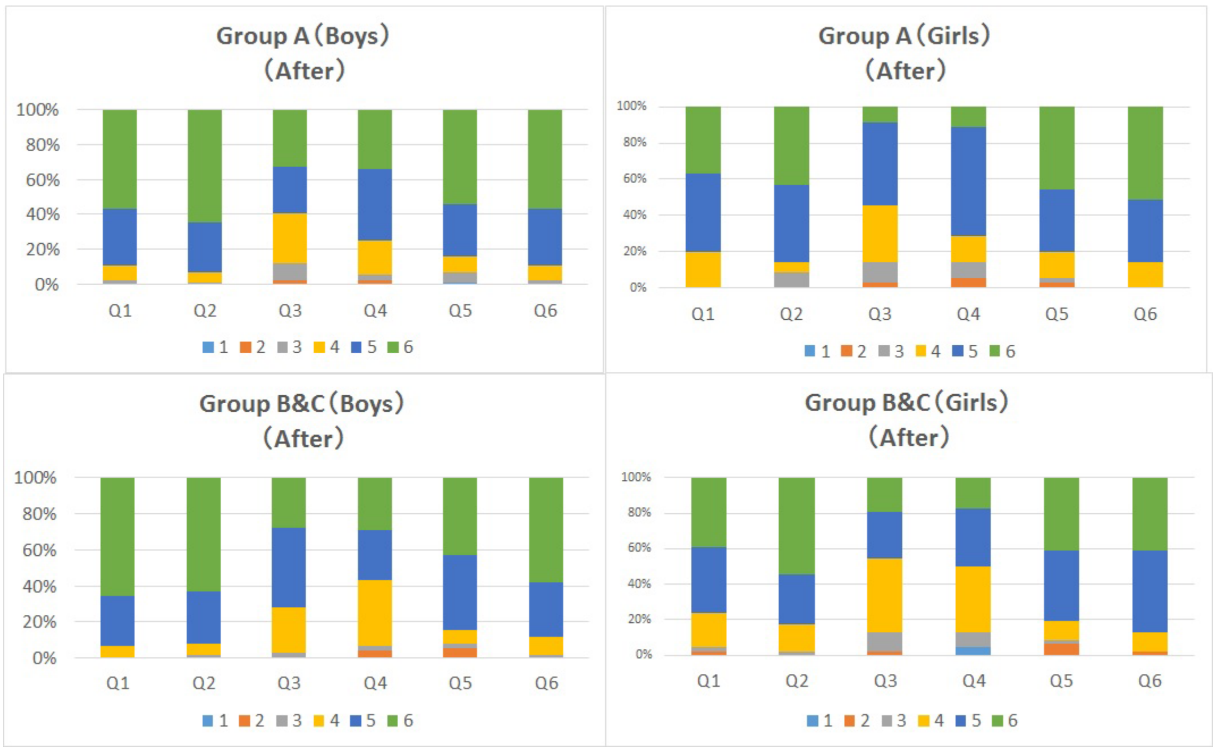


Figure 11: Bar graph of the results of Group A, Groups B&C after experiment according to gender. Color scales denote a rating of 1(strongly disagree)6(strongly agree), respectively. Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness)

average change was 1.082, the girls' was 0.571. It is more ineffective for girl students to employ programmable robots than boy students.

For Q4 (liberal arts) the boys' responses decrease, but the girls' responses increase. As we stated previously, we cannot go into detail about the differences between science and liberal arts majors.

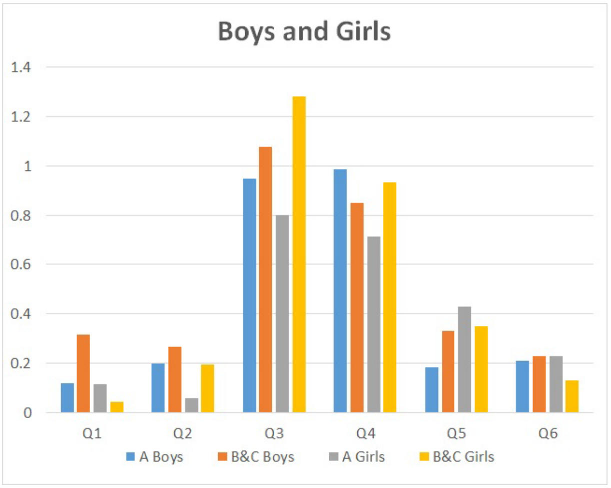


Figure 12: Bar graph of the average of subtraction value. Blue and orange indicate boy students of Group A and Groups B&C, respectively. Gray and yellow indicate girl students of Group A and Groups B&C, respectively.

6.3 Limitation

We analyzed the values of the subtractions using Wilcoxon rank sum test. The results are shown in Table 5.

Question	W	p-value
Q1 (motivation)	6247	0.1309
Q2 (impression)	6377.5	0.2354
Q3 (self-confidence)	6119	0.1031
Q4 (liberal arts)	6994.5	0.9089
Q5 (gender)	6937.5	1
Q6 (usefulness)	6885	0.9082

Table 5: The result of Wilcoxon rank sum test

The p-values of Q1, Q2, Q3, Q4, Q5 and Q6 are 0.1309, 0.2354, 0.1031, 0.9089, 1 and 0.9082, respectively. All of these p-values are larger than 0.05 ($p > 0.05$). There are no statically significant differences in this experiment. However, we do not change our opinions in this research. We think that because there were few scales in this experiment, there are no statically significant differences.

6.4 Threats to Validity

We considered four factors that may influence our findings.

Because we employed questionnaires, the feeling expressed by an adverb such as strongly vs. somewhat in the rating system may vary by individual. Thus, the responses may not be reliable, and our analysis of the motivation to learn programming and the impression of programming may be impacted.

Our experiment only involved middle and high school students. The results may differ if individuals in other age groups participated. Thus, the age of the participants may influence the results.

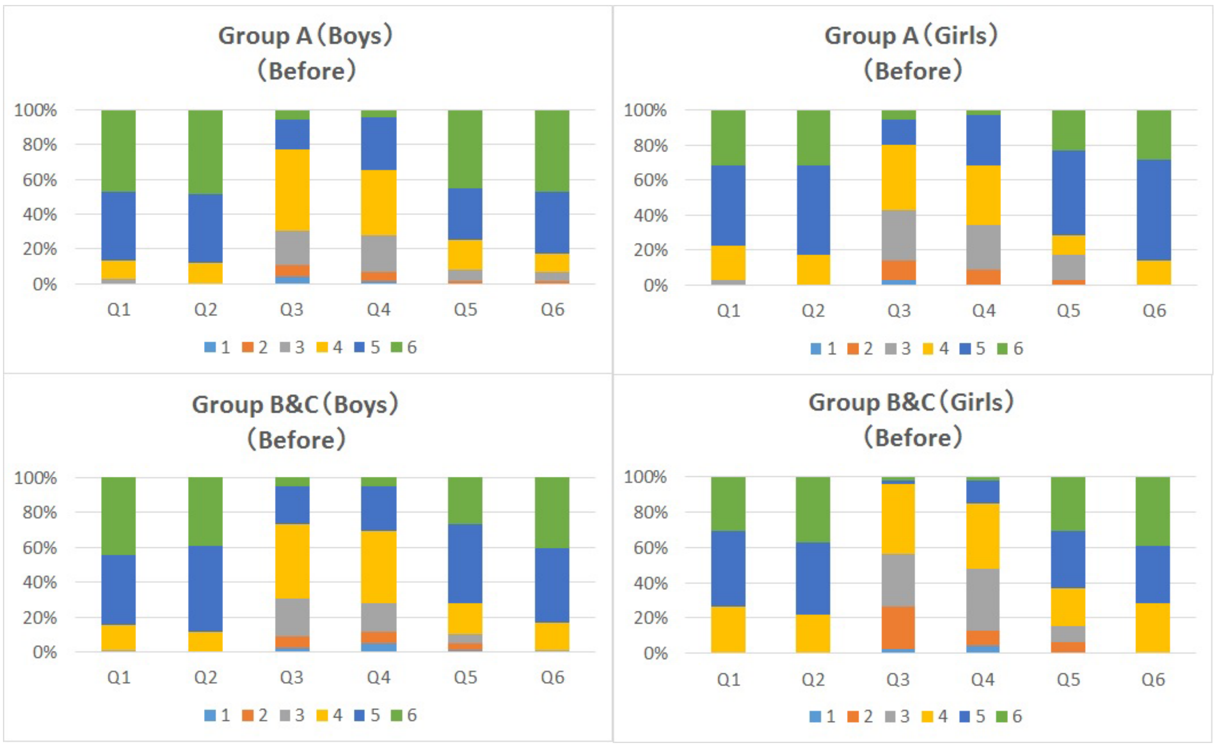


Figure 13: Bar graph of the results of Group A, Groups B&C before experiment according to gender. Color scales denote a rating of 1(strongly disagree)6(strongly agree), respectively. Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness)

Although 236 middle and high school students participated in the experiment, there were only four instructors. Thus, the number of instructors, especially if the student to teacher ratio is one to one, may affect the results.

We randomly divided the 236 students into three groups. Thus, the two scenarios (game-based vs. programmable robot) were not compared using the same student. Thus, a difference in a population may affect the results.

7 Conclusion and Future Work

The contributions of the paper are a large-scale comparative experiment using students learning to program via a game-based application with and without programmable robots. Employing either a game-based application with a programmable robot or without a programmable robot affects the motivation to learn and impression of programming. Additionally, there are gender differences. We answer the following RQs:

RQ1: Does using a game-based application and a programmable robot result in a difference in motivation and impression of learning programming?

RQ2: Compared to a game-based application, does using a programmable robot increase the rate of positive responses to Q1 (motivation), Q2 (impression), Q3 (self-confidence), Q4 (liberal arts), Q5 (gender) and Q6 (usefulness) in the survey?

The answer of RQ1 is that differences exist between using a game-based application with and without a programmable robot. The answer of RQ2 is explained in the following: Using a six items questionnaire, the rates of positive responses to the questions about "motivation" to learn programming, "impression" of programming, "self-confidence" when programming, and ability to program by "gender" increase more when using a game-based application with a programmable robot than when using a game-based application alone. However, the increment in positive responses for questions related to liberal art majors and usefulness is larger for a game-based application alone than a game-based application with a programmable robot. We found that employing programmable robots on learning programming did not always give an improvement to all students. In addition, the rate of positive responses to the questions regarding impression of programming and self-confidence when programming increase for boys, but decrease for girls, while the responses to questions related to programming usefulness and type of major show the opposite trend. It is effective for both boys and girls to employ programmable robots on learning programming for impression and self-confidence only.

Thus, we propose that if you employ programmable robots on learning programming, you can give a good impression and self-confidence of programming, and as for motivation, liberal arts, gender and usefulness, you should take account of the effects depends on students' elements, for example gender.

In the future, we will not only show the effects, especially the motivation to learn and the impressions of programming, but also improve the skills of programming by introducing programmable robots to learn programming. Although we dealt with the problems of a standard difficulty in this experiment, we would like to change the difficulties of the problems to deal with in next experiments. As we mentioned in Section 6.3, we think that because there were few scales in this experiment, there are no statically significant differences. To find statistically significant results, we plan to improve the fineness of the scale and conduct further experiments. In addition, we plan to expand the topics related to learning programming via programmable robots.

Acknowledgments

We would like to thank associate professor Hasegawa of the Tokyo Institute of Technology, Ms. Nishino, Ms. Li, and all of students of the Hasegawa Laboratory for their guidance and assistance in developing the Stuffed Teddy Bear Robot. We would also like to thank Kazuki Otomori, Machiko Hattori, and Yuma Matsuura, who belong to Fukazawa Laboratory and Washizaki Laboratory in Waseda University for their help with our research and the teaching assistants of the open campus at Waseda University. This work was supported by JSPS KAKENHI Grant Number 14504538. This work was also supported by Benesse Corporation.

References

Andersen, P. B., Bennedsen, J., Brandorff, S., Caspersen, M. E. & Mosegaard, J. (2003), 'Teaching programming to liberal arts students: A narrative media approach', *SIGCSE Bull.* **35**(3), 109–113.

Barnes, D. J. (2002), Teaching introductory java through lego mindstorms models, *in* 'Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education', SIGCSE '02, ACM, New York, NY, USA, pp. 147–151.

Bezakova, I., Heliotis, J. E. & Strout, S. P. (2013), Board game strategies in introductory computer science, *in* 'Proceeding of the 44th ACM Technical Symposium on Computer Science Education', SIGCSE '13, ACM, New York, NY, USA, pp. 17–22.

Billard, A., Calinon, S., Dillmann, R. & Schaal, S. (2008), Robot programming by demonstration, *in* B. Siciliano & O. Khatib, eds, 'Springer Handbook of Robotics', Springer Berlin Heidelberg, pp. 1371–1394.

Brennan, K. & Resnick, M. (2012), New frameworks for studying and assessing the development of computational thinking, *in* 'Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada', Citeseer.

Cho, V., Cheng, T. & Lai, W. (2009), 'The role of perceived user-interface design in continued usage intention of self-paced e-learning tools', *Computers & Education* **53**(2), 216–227.

DeClue, T. H. (2003), 'Pair programming and pair trading: Effects on learning and motivation in a cs2 course', *J. Comput. Sci. Coll.* **18**(5), 49–56.

Esper, S., Foster, S. R. & Griswold, W. G. (2013), On the nature of fires and how to spark them when you're not there, *in* 'Proceeding of the 44th ACM Technical Symposium on Computer Science Education', SIGCSE '13, ACM, New York, NY, USA, pp. 305–310.

Fagin, B. S., Merkle, L. D. & Eggers, T. W. (2001), Teaching computer science with robotics using ada/mindstorms 2.0, *in* 'Proceedings of the 2001 Annual ACM SIGAda International Conference on Ada', SIGAda '01, ACM, New York, NY, USA, pp. 73–78.

Feldgen, M. & Clua, O. (2004), Games as a motivation for freshman students learn programming, *in* 'Frontiers in Education, 2004. FIE 2004. 34th Annual', pp. S1H/11–S1H/16 Vol. 3.

Jenkins, T. (2001), The motivation of students of programming, *in* 'Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education', ITiCSE '01, ACM, New York, NY, USA, pp. 53–56.

Kelleher, C., Pausch, R. & Kiesler, S. (2007), Storytelling alice motivates middle school girls to learn computer programming, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', CHI '07, ACM, New York, NY, USA, pp. 1455–1464.

Kölling, M. & Henriksen, P. (2005), 'Game programming in introductory courses with direct state manipulation', *SIGCSE Bull.* **37**(3), 59–63.

Kumar, D. & Meeden, L. (1998), 'A robot laboratory for teaching artificial intelligence', *SIGCSE Bull.* **30**(1), 341–344.

Lalonde, J.-F., Hartley, C. & Nourbakhsh, I. (2006), Mobile robot programming in education, *in* 'Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on', pp. 345–350.

Lewis, C. M. (2010), How programming environment shapes perception, learning and goals: Logo vs. scratch, *in* 'Proceedings of the 41st ACM Technical Symposium on Computer Science Education', SIGCSE '10, ACM, New York, NY, USA, pp. 346–350.

Long, J. (2007), 'Just for fun: Using programming games in software programming training and education', *Journal of Information Technology Education: Research* **6**(1), 279–290.

Magnenat, S., Riedo, F., Bonani, M. & Mondada, F. (2012), A programming workshop using the robot " thymio ii " : The effect on the understanding by children, *in* 'Advanced Robotics and its Social Impacts (ARSO), 2012 IEEE Workshop on', IEEE, pp. 24–29.

Mahmoud, Q. H. (2008), Integrating mobile devices into the computer science curriculum, *in* 'Frontiers in Education Conference, 2008. FIE 2008. 38th Annual', pp. S3E–17–S3E–22.

Malan, D. J. & Leitner, H. H. (2007), 'Scratch for budding computer scientists', *SIGCSE Bull.* **39**(1), 223–227.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M. & Rusk, N. (2008), 'Programming by choice: Urban youth learning programming with scratch', *SIGCSE Bull.* **40**(1), 367–371.

McNally, M., Goldweber, M., Fagin, B. & Klassner, F. (2006), Do lego mindstorms robots have a future in cs education?, *in* 'Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '06, ACM, New York, NY, USA, pp. 61–62.

Nourbakhsh, I. R., Mobile, T., Lab, R. P. & Robots, T. T. (2000), 'Robots and education in the classroom and in the museum: On the study of robots, and robots for study'.

Olivieri, L. M. (2005), ‘High school environments and girls’ interest in computer science’, *SIGCSE Bull.* **37**(2), 85–88.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. & Kafai, Y. (2009), ‘Scratch: Programming for all’, *Commun. ACM* **52**(11), 60–67.

Rizvi, M., Humphries, T., Major, D., Jones, M. & Lauzun, H. (2011), ‘A cs0 course using scratch’, *J. Comput. Sci. Coll.* **26**(3), 19–27.

Sakamoto, K., Takano, K., Washizaki, H. & Fukazawa, Y. (2013), Learning system for computational thinking using appealing user interface with icon-based programming language on smartphones, *in* ‘Proceedings of the 21st International Conference on Computers in Education (ICCE)’.

Takase, Y., Mitake, H., Yamashita, Y. & Hasegawa, S. (2013), Motion generation for the stuffed-toy robot, *in* ‘SICE Annual Conference (SICE), 2013 Proceedings of’, pp. 213–217.

Wing, J. M. (2006), ‘Computational thinking’, *Commun. ACM* **49**(3), 33–35.

Mired in the Web: Vignettes from Charlotte and Other Novice Programmers

Donna Teague
Queensland University of Technology
Brisbane, QLD, Australia
d.teague@qut.edu.au

Raymond Lister and Alireza Ahadi
University of Technology, Sydney
Sydney, NSW, Australia
Raymond.Lister@uts.edu.au

Abstract

Ahadi and Lister (2013) found that many of their introductory programming students had fallen behind as early as week 3 of semester, and those students often then stayed behind. Our later work (Ahadi, Lister and Teague 2014) supported that finding, for students at another institution. In this paper, we go one step further than those earlier studies by observing a number of students as they complete programming tasks while thinking aloud. We describe the types of inconsistencies students manifest, which are often not evident on analysis of conventional written tests. We again interpret our findings using neo-Piagetian theory. We conclude with some thoughts on the pedagogical implications of our research results.

Keywords: Programming, neo-Piagetian theory, novices, assessment, think aloud.

1 Introduction

Many computing educators have noted a large variation in the ability of introductory programming students. Ahadi and Lister (2013) found significant differences in performance among their students, as early as week 3, on trivial coding problems. Furthermore, those students with lower scores on the week 3 test also tended to perform lower on test questions in subsequent weeks — that is, some students fall behind very early and then stay behind.

Ahadi et al. (2014) conducted a second study, comparing students at two different institutions. They found that tests held early in semester were good indicators of success in the final exam. In this paper, we report on a similar quantitative study, but we go further, by triangulating with qualitative think aloud data from students completing the same test questions.

2 Neo-Piagetian Theory

Lister (2011) proposed that we can describe students' development in programming in terms of neo-Piagetian theory. Other studies (Falkner, Vivian, and Falkner 2013; Teague et al. 2013; Teague and Lister 2014c) provide empirical evidence of novices manifesting neo-Piagetian stage-related characteristics as they reason about programming tasks. According to the evidence accumulated from these and related studies, the first three stages of development are characterised as follows.

At the **sensorimotor** stage, novices tend to inconsistently apply mis/conceptions about programming. Because of their fragile knowledge, these students struggle to successfully trace code, let alone reason about its purpose or write their own code.

At the next more mature level are **preoperational** students who have begun mastering the semantics, and any misconceptions that remain at this stage are at least applied consistently. Although preoperational students can accurately trace code, they are often not able to reason about its purpose other than by induction from input/output pairs (see Teague and Lister (2014b)).

It is at the **concrete** operational stage, the next more mature stage, where students have developed an ability to reason deductively about abstractions and write more complex code. This is the stage at which computing educators typically expect students to be working by the end of their first semester of learning programming, and the level at which students are traditionally assessed. However, the findings of this study, and previous studies**,** suggest that many students are not manifesting concrete operational skills even by their second semester of study (Teague et al. 2013).

Rather than making quantum leaps between these three stages, our view of development is described by the Overlapping Waves Model (Boom 2004; Feldman 2004; Siegler 1996). In that model, characteristics of an earlier stage dominate initially, but there is a gradual increase in the use of the next more mature level of reasoning and a decrease in the less mature stage. This model accounts for students manifesting characteristics of more than one stage simultaneously.

3 Method

The undergraduate introductory programming course we studied ran at the first author's institution over a 13 week semester comprised of a two hour lecture and a two hour workshop each week.

To collect the data for this study, students completed a short "in-class" test at the start of the lectures in weeks 2, 4, 7 and 9. These tests did not contribute to a student's final grade. However, most students present at the lecture did the test, as the lecture did not proceed until the test was over. The time students took to complete a test was not formally recorded, but each test took around 15 minutes. Students were under little time pressure. Immediately after each test, the lecturer would review the test and explain the correct answers.

Much of the work of the first author in recent years has involved observing approximately 40 individual student programmers, as they developed over the course of a semester. Those students completed programming tasks while thinking out loud (Ericsson and Simon 1993). In this paper we describe some of those students' attempts at the tasks that in-class test data identified as being problematic for many students. The qualitative data from the think aloud sessions help to answer some of the questions that arise from the in-class results:

What strategies do students use? (In other words, how did they get that answer?*)*;

What behaviour is evident with students who have difficulty completing programming tasks?; and

What programming misconceptions (if any) are evident? (Are incorrect answers a result of careless mistakes, misinterpretation of the question or lack of understanding the concept?)

Once we have that information, we can answer the "why" questions by interpreting the qualitative data using the neo-Piagetian framework:

Why do students get particular questions wrong?

Can a student have disparate levels of ability with two tasks which test similar programming concepts? (For example tracing, explaining and writing the same code.)

Why are some students unable to work with abstractions? (For example, why do they rely on tracing code with specific values?)

It is not possible to include all our think aloud data in this paper. We have simply selected three sessions that are representative of the broadly different types of reasoning manifested by our think aloud students.

We use aliases to obfuscate students' identity. Excerpts from the sessions with Charlotte ("C"), Lance ("L) and Jim ("J") are detailed in the following sections. Lance was in the same cohort as those completing the in-class tests. Unlike the others, Charlotte was a postgraduate student, but as she was in her first programming unit at the time of her think aloud session, she was at a similar level to those students in the in-class tests. Jim was in week 2 of his second programming unit.

In these excerpts, a pause in speech is marked "...", as a placeholder for dialog we have removed as it added nothing to the context of the think aloud session.

4 Test 1 (Week 2)

When the students completed Test 1 at the beginning of their week 2 lecture, they had completed two hours of lectures and a two hour workshop. The test questions are provided in the appendix. (We will hereafter refer to test questions in an abbreviated form. For example, Question 1 will now simply be Q1.) Our Test 1 is very similar to the Test 1 of Ahadi and Lister (2013), differing in only four respects: (a) our test is a translation from their Java to our Python, which is a trivial change given that all the questions in Test 1 are about assignment statements; (b) we renumbered their questions, (c) we omitted Q2a from the Ahadi and Lister test, but retained their Q2b as our Q7; and (d) we conducted our first test in week 2 whereas they conducted their first test in week 3.

Figure 1 shows the distribution of student scores on Test 1, where 8 is the maximum possible score.

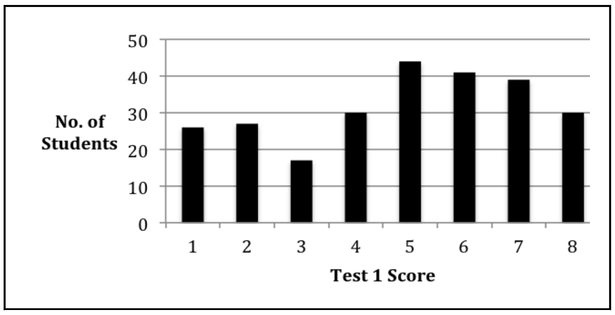


Figure 1: Distribution of total scores on Test 1 (N=254)

All questions were worth 1 point, with no fractional points awarded. Answers were treated as either right or wrong, but syntactic errors were ignored. We eliminated from Figure 1 and all subsequent analysis, the small number of students who scored zero on Test 1, as they were likely to be students who had not attended week 1 classes. As was the case for Ahadi and Lister (2013), there was a wide variation in Test 1 scores.

Table 1 shows the percentage of students, for each Test 1 score out of 8, who correctly answered each of the eight questions. The final row of the table represents the percentages of all students who answered correctly each question in the test. Cells containing asterisk/s indicate a statistically significant difference in the two percentages above and below the asterisk/s. (NB: percentages are rounded down.) As can be seen from that table (especially for test scores of 1 to 6 inclusive, as marked with darker border lines), an approximate rule of thumb is that if a student scored *n* points out of 8 on the test, then the student's first *n* answers were most commonly right, and their remaining answers were most commonly wrong. In accordance with that rule of thumb, we characterised the students as follows:

- Score 1 or 2: understands little of the semantics of the code.
- Score 3 or 4: applies inconsistent guessing because of fragile understanding of the semantics.
- Score 5: can conduct a trace with some reliability.
- Score 6: can perform inductive inference.
- Score 7: can sometimes perform deductive inference.

We elaborate on this characterisation in the next section.

4.1 Semantics of Assignment and Sequence

In Test 1, Q1–Q3 tested whether a student understood the semantics of a sequence of assignment statements. That is, the value on the right of the assignment is copied to the left, overwriting the previous value, and assignments are executed in sequence. Many students who scored 1, 2 or 3 on Test 1 struggled with Q1–Q3 (see the left three shaded columns in Table 1).

Test1 Score	*n*	semantics			tracing		reasoning		
		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	26	53	23	0	4	4	8	12	0
			**	**	*				
2	27	60	71	26	26	15	4	0	0
				***	*				
3	17	53	65	89	59	24	6	6	0
		*				*			
4	30	87	84	80	64	54	14	14	7
			*		**	**			
5	44	87	96	94	94	85	30	12	5
							***	*	***
6	41	86	98	98	96	88	69	35	35
								***	***
7	39	83	100	98	93	98	75	75	80
		*					**	**	**
8	30	100	100	100	100	100	100	100	100
all	254	78	84	76	73	65	43	34	32

Table 1: Percentage of students who answered correctly each part of Test 1, broken down by total score (χ^2, * is $p \leq 0.05$, ** is $p \leq 0.01$ and * is $p \leq 0.001$)**

Sensorimotor students often have no alternative but to use guessing as a strategy for reasoning about code. This is because they have not yet built a clear mental model of the notional machine (du Boulay 1989), nor do they have a solid comprehension of the concepts to which they have only just been introduced. Because of this, they inconsistently apply mis/conceptions about the semantics of code.

4.1.1 Vignettes from Charlotte

One of our think-aloud students, Charlotte, demonstrated this type of sensorimotor reasoning when she was asked to trace the effect of the three assignment statements (Q2) shown both in the appendix and again here in Figure 2.

```
r = 2
s = 4
r = s
```

Solution: `r` is 4, `s` is 4

Figure 2: Test 1 Q2 - Tracing Task

As Charlotte considered the code she said:

C: *Hmm. ... I don't know, but I imagine ... it's kind of a guess here [laugh], that ...* `r` *will equal 4 ... and* `s` *will equal 4.*

Of course students will get the marks for correct guesses in exams, and as this think aloud session showed, it is not until you listen to a student's reasoning that you can start to understand their true level of ability. This is consistent with the findings of Teague et al. (2012) who provided an astonishing contrast between the correct solution a programming student was able to produce and the inexplicable reasoning and method he actually used to produce that solution. This is of course the advantage of think alouds. It is quite obvious when a student flukes a correct answer. Think alouds also explain why, in other cases, students answer incorrectly.

With her very next task (Q3, shown again in Figure 3), Charlotte thought she was being consistent with her "guess", but that was not actually the case.

C: *So…going from how I did the last one, I might as well be consistent. ...* `p` *will equal 8 and* `q` *will equal 1.*

```
p = 1
q = 8

q = p
p = q
```

Solution: `p` is 1, `q` is 1

Figure 3: Test 1 Q3 – Tracing Task

Charlotte later reflected on that answer and explained:

C: *I looked up to the original integer rather than looking at the switched integer*

In other words, she looked only to the first assignment of `q` (i.e., `q = 8`) rather than taking account of its subsequent reassignment (`q = p`). Charlotte's fragile understanding of the semantics (as well as a floundering command of the jargon) is also exemplified in her next comment:

C: *I'm just not confident in how the rules of inheritance were applied. It was like I was just going on a whim.*

Students who scored 4 on Test 1 tended to answer Q1–Q3 correctly, and *either* Q4 or Q5 correctly. We characterise these students as novices who still have a fragile understanding of the semantics of the language, and like Charlotte, inconsistently apply mis/conceptions.

4.2 Inductive Reasoning

Lister (2011) proposed that a preoperational programming student can make reasonable inductive guesses about the function of a piece of code based upon the input/output behaviour they observe from tracing it, without understanding how the code achieves that function.

We have witnessed this type of reasoning in previous work (Teague et. al. 2013, Teague and Lister 2014a) where the student (Donald) attempted to explain the purpose of code that sorted the values in three variables. Donald based his answer on the effect of a single set of poorly chosen input values. As a result, his answer, although accurate for that single test case, did not reflect the purpose of the code for *any* set of input values.

The students described in this paper who scored 5 on Test 1 usually answered all the tracing questions correctly (Q1–Q5) but often could not explain the swap code they had just traced (Q6). In fact, Table 1 shows that out of the students who scored 5 on the test, only 12% of them could *explain* similar swap code (Q7); and only 5% of them could *write* similar swap code (Q8).

4.2.1 More Vignettes from Charlotte

Charlotte is illustrative of those students who can sometimes trace a piece of code but cannot explain that code. In her previous two tasks, Charlotte guessed, and applied inconsistently her misconceptions about assignment statements. It is not surprising, therefore, that her ability to reason about the *purpose* of code (Q6, shown in Figure 4) is very limited. This time, Charlotte traced the code accurately (or at least managed to guess the correct effect of assignment consistently), but she was unable to explain the code's overall purpose:

```
x = 7
y = 5
z = 0

z = x
x = y
y = z
```

Solution: The values in `x` and `y` were swapped

Figure 4: Q6 – Reasoning Task

C: *So if* `z` *equals* `x` *from above, that will become 7 ... If* `x` *becomes* `y` *...* `y` *is 5, so* `x` *becomes ... 5 ... If* `y` *equals* `z` *... it becomes 7, so I don't know what I observe.*

As shown in Table 1, of the students who scored 6 on Test 1, approximately two thirds of them made the correct observation for Q6, but only about a third could answer either Q7 or Q8.

Table 2 shows contingency tables for Q6 and Q7, and also Q6 and Q8, for those students who answered both Q4 and Q5 correctly. Most students who answered Q6 (explain swap by induction) incorrectly could not answer correctly either Q7 (explain swap by deduction) or Q8 (write swap). Even among students who did answer Q6 correctly, a substantial percentage could not answer correctly either Q7 or Q8. As a rough guide, answering Q6 correctly tends to be a necessary, but not sufficient, condition for answering Q7 and Q8 correctly.

Test 1 Q6 "what do you observe about final values in x and y" (induction)	Test 1 Q7 "explain swap" (deduction)		Test 1 Q8 "write swap"	
	wrong	right	wrong	right
wrong (n = 55)	26%	13%	30%	9%
right (n = 89)	28%	33%	25%	36%

Table 2: Contingency tables for Q6 & Q7 and Q6 & Q8, for students who answered both Q4 & Q5 correctly (χ^2, p= 0.012 for Q7 and p < 0.001 for Q8, N=144 for each of Q7 & Q8)

As noted above, Charlotte was one of those students who could not answer Q6 correctly. She was prompted by the interviewer to see that the code was swapping the values in variables `x` and `y`. She was then asked to explain the Q7 swap code, shown in Figure 5.

```
j = i
i = k
k = j
```

Solution: The values in `i` and `k` were swapped

Figure 5: Q7 – Reasoning Task

C: *when these lines of code are executed,* `j` *becomes ... is already* `i`. `i` *is* `k`, `k` *is* `j`, *so thereby ...* `j` *equals* `k` *which is already done at the end so I doubt that's right*

Perhaps Charlotte was reading the "=" as a statement of mathematical equality: if `j` is equal to `i`, and `i` is equal to `k`, then `j` is equal to `k`. However, the "=" operator is about assignment, not equality. In any event, Charlotte then shifted her reasoning about the code from being about statements of equality, to assigning values:

C: *Oh, well maybe ...* `j` *equals* `i`, `i` *equals* `k`, `k` *equals* `j` *...Yeah! well it takes away the need for* `i`.

Our interpretation of what Charlotte said is that `i` is not needed when swapping the values in `j` and `k`. In other words, a swap can be effected simply by assigning `k` to `j` and then `j` to `k`. Whatever her reasoning, we have seen that it is confused.

4.3 Deductive Reasoning and Code Writing

Lister (2011) proposed that deductive reasoning in programming was the ability to infer the computation performed by a piece of code, without needing to trace the code with specific values. Such ability is characteristic of the concrete operational stage in neo-Piagetian terms.

Students who scored 7 on Test 1 tended to answer all the tracing questions correctly (i.e. Q1–Q5) but tended to only answer correctly two questions out of Q6, Q7 and Q8, in near-equal percentages (75%, 75% and 80% respectively).

Table 3 shows the relationship between Q7 (explain swap by deduction) and Q8 (write swap) among the 144 students tested. Among these students, 24% of them could only answer one but not both of Q7 and Q8 correctly. However, a greater percentage of students who had explained the swap (Q7) could write a swap (Q8). This result is consistent with earlier findings by others that the ability to explain code is a prerequisite for the ability to write similar code (Lopez, Whalley, Robbins, and Lister 2008).

Test 1 Q7 "explain swap"	Test 1 Q8 "write swap"	
	wrong	right
wrong (n= 79)	43%	12%
right (n = 65)	12%	33%

Table 3: A contingency table comparing the performance of students on Q7 and Q8, for the students who answered both Q4 and Q5 correctly. (χ2, p < 0.001, N = 144)

4.3.1 Vignettes from Jim

Jim, another think-aloud student, had trouble with both Q7 and Q8, even after completing Q1–Q6 successfully. Jim looked at the code in Q7 (see Figure 5) and said:

J: `j` *has been changed ... to take the value of* `i` *... because* `j` *took the value of* `i`, *so* `k` *takes the value of* `j` *... therefore* `k` *is taking the value ... of* `i` *...*

Here, Jim used only the first and third lines of code in Figure 5 (and ignored the second line where `i` is reassigned) to reason about the value being assigned to `k`.

J: *so it's just a loop.*

By "loop" we believe Jim meant something about the movement of data between the variables rather than a looping control structure in the code. Jim's misconceptions about the assignments remained evident when he then took into account the second line of code, having considered the code in order of lines 1, 3 then 2:

J: So ... basically `k` *will keep its value and everything will become the value of* `k`.

In other words, his reasoning was: `j` is given the value of `i` (line 1); therefore `k` (in line 3) is taking the value of `i` too because it is assigned `j`; and `i`'s value originally came from `k`. So therefore, `k` is unchanged by this process, and the other variables both have the value of `k`. After the interviewer questioned Jim's summation (i.e. that `k` remained unchanged) he became less sure:

J: *No, the `k` will keep it's ... `j` will keep its value... no*

By this stage, Jim was confused and probably cognitively overloaded. He decided to restart the task and this time he wrote specific values for each of the variables. Resorting to tracing with specific values is typical behaviour for students who are yet to reach the concrete operational stage and who are weak at reasoning with abstractions.

J: *Ok, we'll just say ... we have `j` is equal to 1, `i` is equal to 2 and `k` is equal to 3.*

Jim traced the code again using those specific values which he wrote above the variables. However, he made a transposing error with the final line, causing him to assign `k`'s value to `j` instead of the other way around. His final trace of the three lines of code in Q7 (Figure 5) is shown in Figure 6.

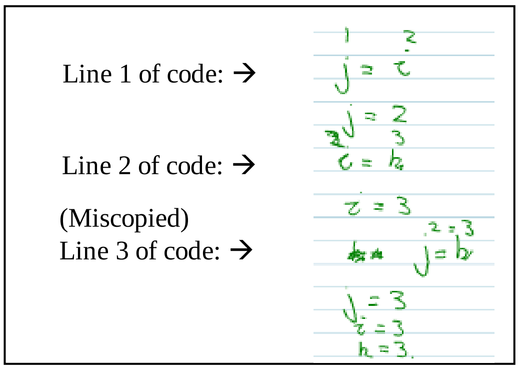

Line 1 of code: →
Line 2 of code: →
(Miscopied)
Line 3 of code: →
1 2
j = i
j = 2
3
i = k
i = 3
2 3
j = k
j = 3
i = 3
k = 3

Figure 6: Jim's trace of Q7

Jim was prompted to recheck this trace, and the interviewer suggested that a clearer way to articulate assignment was to say "is given" (rather than "is equal to") to help him focus on the direction of the assignment. Jim then corrected the miscopied assignment statement at line 3 in Figure 6 (to:"`k = j`"), but said:

J: *`k` is given to `j`, there we go*

Jim seemed to be getting confused between the direction of assignment (i.e. the movement between variables) and the articulation of the assignment statement (i.e. reading left to right). So the interviewer ("I") intervened further:

I: *No. `k` is assigned the value of `j`. So `j` is given to `k`. Depends which way you want to read it. ...*

J: *Yeah, so ... `j` becomes `k`.*

I*: No. in this case, k becomes j*

J: *oh, `k` becomes `j` sorry ... so `k` is equal to 2.*

Given the difficulties with assignment that Jim manifested here in Q7, it is surprising that Jim managed to answer Q1 to Q6 correctly. We speculate that Jim's problems here are due to the higher cognitive load.

Finally having traced the code correctly, Jim attempted to explain its purpose. This proved even more difficult:

J: *it's just really reassigning. Isn't it? Because we have `j` is equal to 2, `i` is equal to ... 3 and `k` is equal to 2.*

Jim's response is a vague overview of the code, equivalent to "all the variables have been changed". Asked if the code was doing something similar to that in the example in Q7 he replied:

J: *it's similar, in the sense that it's swapping ... um, we've got .... `c` becomes `a` ... `a` becomes ... `b` and `b` becomes `c`, so that's just swapping them*

In terms of the SOLO taxonomy (Biggs and Collis 1982) this is a multistructural answer – recounting the effect of each individual line, rather than the total effect of all three lines. Asked which variables are swapped:

J: *the first ones ... `j` swapped, `j` took the value of `i` ... `i` and `j` swapped*

It is clear now that what Jim meant by "swap" was "change", rather than a two-way exchange of values. After clarification of what a "swap" was, and looking at what each of the variables started and ended up with, Jim was finally able to answer that indeed there had been a swap of values between two variables:

J: *apparently `i` swapped with `k`*

Jim's use of the word "apparently" suggests a lack of conviction. His difficulty with the tracing task showed misconceptions which are characteristic of novices at the sensorimotor stage. However, sensorimotor novices are also reluctant to retrace as it is a cognitively demanding task given their fragile domain knowledge. But Jim decided to redo the task, this time in a manner he was more comfortable with. He introduced specific values. Novices at the preoperational stage are unable to deal solely with abstractions and require specific values to make sense of code. In terms of the Overlapping Waves Model (as described Section 2), we suggest that Jim is in the process of developing preoperational skills, while still displaying some legacies of the sensorimotor stage.

4.3.2 Vignettes from Lance

After seeing how Jim dealt with reasoning about three lines of assignment statements, the reader will not be surprised that he had difficulty writing similar code. In fact (as shown in Table 1) 20% of the students who scored 7 correctly answered all of preceding tracing and reasoning questions (Q1–Q7) but then could not *write* similar code (Q8).

Our final think aloud student, Lance, had difficulty writing the code, even though he had answered Q1–Q7 correctly. For Q8, Lance wrote the first (correct) lines of code to swap the variables `first` and `second`:

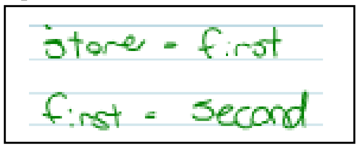
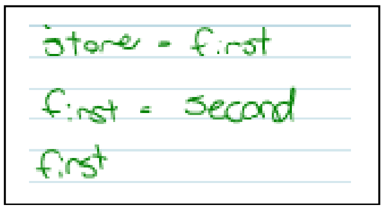

store = first
first = second

Figure 7: Lance's 1st two Lines of Q8 Swap Code

But his explanation of that code was inaccurate:

L: *ok so now ...* `second` *should have the number that* `first` *has in it*

Lance had written the assignment statement in one direction and articulated it in the opposite direction. He continued with the third line of code before hesitating:

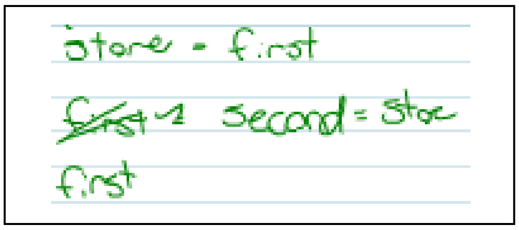


store = first
first = second
first

Figure 8: Lance's 3rd Line of Q8 Swap Code

L: *oh no that's wrong ... I think ... that* <u>*is*</u> *wrong because ... um ... ok it should be* `second` *equals* `store` *... shouldn't it*

Lance changed his code to:



store = first
~~first =~~ second = store
first

Figure 9: Lance's Revised 2nd Line of Q8 Swap Code

After reading his revised code, Lance decided to start again. Like Jim and other novices reasoning at the preoperational stage, this time he used specific values to help him reason about the code he was writing.

L: *ok so you've got ... let's just say that's 1 and that's 2 so I can keep it in my head. ok this will make it a bit easier alright*

While Lance assigned the values 1 and 2 to variables `first` and `second`, writing the code still proved not to be straight forward:

L: *so first you're going to need to store the ... memory of* `first` *... like the number in* `first` *... so we're gunna go ...* `store` *... equals* `first` *...*

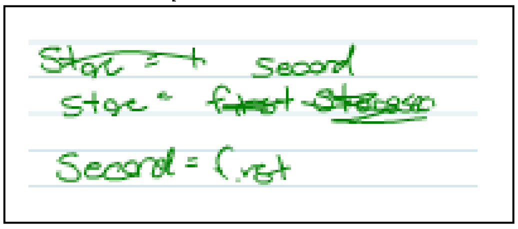
Although Lance said "`store` ... equals `first`" he wrote "`store = 1`". We don't believe he meant to write "1", but he was no doubt thinking that `first` had the value 1. He was working at the preoperational level at which it is difficult to reason in abstract terms. In any event, he quickly self-corrected this error by changing the code to "`store = first`".

Lance then gave an confused explanation of what the code needed to do:

L: *ok ... just stored ... the number from* `first` *into ...* `store` *... then you go from ... we need to put the number that* <u>*was*</u> *in* `first` *into* `second` *so if we go ... because we're stored* `first` *we can put ... that in there because it's remembered now ... so if we go ...* `first` *equals* `second` *... I think ... no that's what I was doing before ... and I thought it was wrong ... maybe if we just store* `second`

Lance sought confirmation from the interviewer that it would in fact make no difference whether he stored the value of `first` or `second` to begin with. He decided to make the change anyway, although he wrote by mistake "`store = stores`". After fixing this error he said:

L: *ok so* `store` *equals* `second` *... why is it so confusing it's so simple [laugh] confusing ... alright* `store` *equals* `second` *so you go* `store second` *and then ... that number's remembered ... and that's 2 ... and basically we want to assign that ... to ... we want to assign* `first` *... alright we want to overwrite the 2 in* `second` *... to the 1 in* `first` *so if we go ... um ...* `second` *equals* `first`



Store = ~~1~~ Second
store = ~~first~~ ~~store~~second
Second = first

Figure 10: Lance's 2nd Attempt at Q8 Swap Code

Although he made no note of the changing values on paper, Lance constantly used specific values to talk about the effect of the assignments. He seemed unable to cope with even the abstraction of variable names. As he said before, using specific values makes it easier for him "to keep in his head". And this tactic eventually worked.

L: *so now you've got ... ah the 1 in* `second` *... and the 2 in* `store` *and then if you go* `first` *equals* `store`*...*

In summary, when it came to writing code in Q8, Lance struggled to implement code very similar to code he had just successfully traced and reasoned about. He failed to write code until he introduced specific values, which enabled him to visualise the changing values in the variables. Preoperational novices are reliant on specific values to reason about and write code.

Only 30 students (12%) who completed Test 1 scored the maximum possible 8 marks, and were deemed competent at tracing, reasoning about and writing very simple code. Given their consistent correct performance, these students are unlikely to have been guessing about the semantics of the code. The fact that they were also able to write the code in Q8 would lend us to believe that they were at least operating at the preoperational level. While these students may be reasoning at the concrete operational stage we are reluctant to draw that conclusion with confidence, without knowing how they went about solving the problems, given the evidence of superficially correct solutions presented by Teague et al (2012).

5 Test 2 (Week 4)

We conducted our second test two weeks later, in week 4.

5.1 Test 2 Q1 (tracing question)

This first question in Test 2 was a tracing question equivalent to the last tracing question in Test 1 (Q4). Students who scored 1–4 in Test 1 tended to perform poorly on the last tracing question in that same test (Q4, see Table 1). However, all students performed very well on the first tracing question in Test 2, with the probability of answering this question at 77% for those who scored 2 in Test 1, and at 96% for all other students. So the students who had lagged behind on tracing skills in week 2 had substantially closed the gap by week 4, at least on this type of question.

5.2 Test 2 Q2 (writing question)

The second question in Test 2 was exactly the same as Q8 in Test 1. That is, the students were required to write code to swap the values in two variables, `first` and `second` (see appendix).

Figure 11 plots the probability of students answering this Test 2 question correctly, against their total score on Test 1. The largest circle in Figure 11 represents 26 students, while the smallest circle represents 10 students.

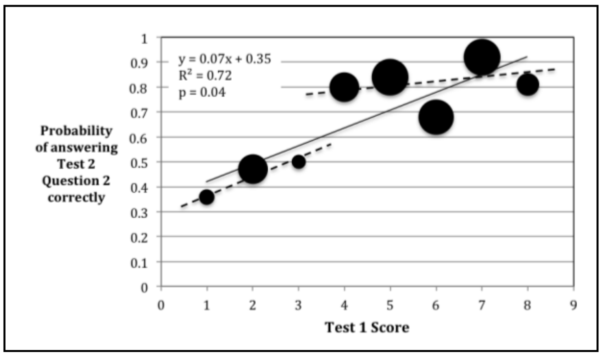


Figure 11: Relationship between Test 1 scores and the probability of answering Test 2Q2 correctly (N=156)

The solid regression line shown in Figure 11 accounts for 72% of the variation, and that regression line is statistically significant at the 0.05 level. Therefore overall performance on Test 1 (week 2) is a good predictor of performance on this code writing question in the week 4 test (Test 2, Q2). Recall from the previous subsection, however, that performance on the week 2 test was not a good predictor of performance on the week 4 tracing question (Q1), so we cannot conclude simply that students who do better on Test 1 tend to do better on all questions in subsequent tests.

Inspection of Figure 11 suggests that, although the solid line of regression is a good predictor, there does appear to be a non-linear jump in performance between students who scored 1–3 on Test 1 and students who scored 4–8. The two dashed lines are lines of regression through each of those two groups of students, and serve to highlight that possible performance gap. Note, however, that neither of these two dashed regression lines meets the traditional 0.05 statistical criterion for significance, perhaps because of the small sample size. This possible performance gap suggests that, while students who scored 1–3 on Test 1 have closed the gap on tracing skills for these simple tracing problems, they have not closed the gap on deductive and code writing skills. That is, while students who scored 1–3 on Test 1 are progressing in their learning, they are not progressing as quickly as students who scored higher on Test 1. Our interpretation of this in neo-Piagetian terms is that the students who scored 1–3 on Test 1 were now better at tracing code, but they were still operating (at most) at a preoperational level of reasoning. They had not made the transition to the concrete operational stage. They remained unable to reason about abstractions and therefore unable to write simple code.

6 Test 3 (Week 7)

Our third test was conducted in week 7, five weeks after the first test. By this stage of semester, students had been introduced, amongst other concepts, to conditional statements and Python lists.

6.1 Test 3 Q1 (swapping list elements)

Figure 12 shows the first question from Test 3, which also requires students to write a swap, but in this case it is a swap between two elements of a Python list.

A list called `ages` has been created in Python. There are two values out of order in the list and these values are stored at indexes 0 and 2. Write code to swap those two values so that the list would be in order.

Sample Solution:

```
        temp = ages[0]
        ages[0] = ages[2]
        ages[2] = temp
```

Figure 12: Test 3 Q1 with sample solution

Figure 13 plots the probability of students answering Test 3 Q1 correctly, against their total score on Test 1. The largest circle in Figure 13 represents 18 students, while the smallest circle represents 4 students.

While the regression in Figure 13 does show a statistically significant linear relationship ($p < 0.01$), there is a clear non-linearity in the neighbourhood of the Test 1 score of 5. A non-parametric χ^2 test shows that the gap between scores of 5 and 6 is statistically significant at the 0.1 level (see Table 4).

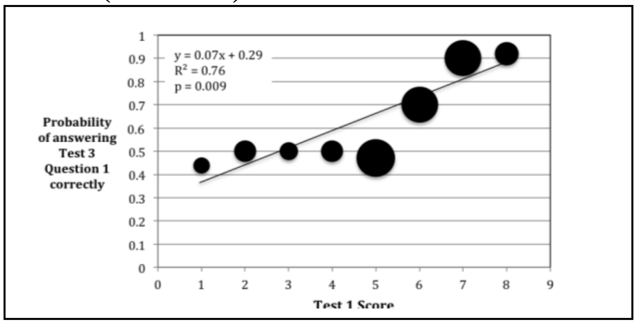


Figure 13: Relationship between Test 1 scores and the probability of answering Test 3 Q1 correctly (N=117)

Thus students who could not perform inductive inference (i.e. those operating at the sensorimotor level) in the week 2 test are, 5 weeks later, still tending to reason at the sensorimotor level, and lag behind those students who could perform inductive inference (i.e., those operating at least at the preoperational level) in week 2.

Test 1 score	N	Test 3 Q1	
		Wrong	Right
5 (i.e. typically could trace with some reliability in Test 1)	21	52%	48%
6 (i.e. typically could perform inductive inference in Test 1)	20	30%	70%

Table 4: A contingency table comparing students on Test 1 scores 5 & 6 versus Test 3 Q1 (χ^2, p=0.1, N=41)

The gap between Test 1 scores of 6 and 7 is also statistically significant at the 0.1 level (see Table 5).

Students who could not perform deductive inference (at best, preoperational) in the week 2 test are, 5 weeks later, still lagging behind those students who could perform deductive inference (concrete operational) in week 2.

Test 1 score	N	Test 3 Q1	
		Wrong	Right
6 (i.e. typically could perform inductive inference in Test 1)	20	30%	70%
7 (i.e. could sometimes perform deductive inference in Test 1)	20	10%	90%

Table 5: A contingency table comparing students on Test 1 scores 6 & 7 versus Test 3 Q1 (χ^2, p=0.1, N=40)

7 Test 4 (Week 9)

We conducted a final test in week 9. One of the questions required students to write code to swap values in a list. On this occasion the values in the list were to be swapped only if they were out of order. The only students who did well with this question were those who scored 100% on Test 1. For all other students, the probability of getting it right was less than 50%.

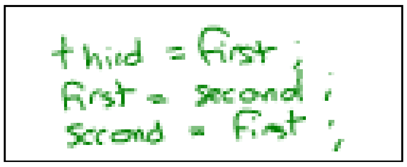
Among those who scored 1 to 7, there appears to be a performance gap on this question with students who performed very poorly on Test 1 (29% probability for Test 1 scores 1–3) performing considerably worse than the students who demonstrated some ability to trace reliably in Test 1 (49% for scores 4–7).

8 Charlotte's Progress

We have so far seen that Charlotte struggled in Test 1 to both trace and explain simple assignment statements. In neo-Piagetian terms this means she was likely reasoning at the sensorimotor stage. Not surprisingly, she also failed the concrete operational task of code writing in that same test. She hypothesised that a third variable would be required in order to make a swap, referring to the code shown in the previous question (Test 1 Q7, see appendix).

C: I'll follow the format from above ... 'cause it makes sense 'cause it worked

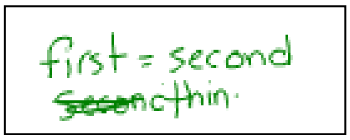
Her strategy was to give each of the variables a value, and she noted what their values should be once her code had executed. Then she wrote the incorrect code in Figure 14.



```
third = first ;
first = second ;
second = first ;
```

Figure 14: Charlotte's First Attempt in Week 2

When Charlotte attempted the very same code-writing task five weeks after her first think aloud, she still struggled with it. She initially failed to use a third (temporary) variable, as can be seen from the first line of code in Figure 15. For the second line, she started writing "second", crossed it out and replaced it with (an incomplete) "third" before crossing out all that she had written (shown in Figure 15).



```
first = second
~~second~~thir
```

Figure 15: Charlotte's Second Attempt in Week 7

Charlotte almost immediately then wrote correct code, and verified her solution using specific values for `first` and `second`. Charlotte was now, five weeks after the first think aloud, working at the preoperational level: having overcome her initial misconceptions, she was able to trace and write very simple, familiar, code.

Two weeks later, Charlotte completed Test 4 before we had a think aloud session with her. Her final code for a conditional swap of list elements was accurate. However, when she reflected on this question in a subsequent think aloud session, Charlotte confessed to not being sure of the correctness of her solution and voiced some confusion about assigning array elements:

C: I was thinking `temp` had to be an array…

Having previously developed the ability write swap code, Charlotte was then manifesting misconceptions with less familiar material: arrays. Her behaviour is consistent with an Overlapping Waves Model, where the introduction of a new concept may result in reversion to a less mature stage (for that concept).

9 Conclusion

Our think aloud excerpts have answered the first of the questions posed earlier, regarding the strategies, behaviour and misconceptions that are evident in novice programmers. We categories these (in Table 6) using the neo-Piagetian (NP) framework (where SM=sensorimotor; Preop=preoperational).

Behaviour	**NP Stage**
guessing	SM
fragile grasp of semantics	SM
confused use of nomenclature	SM
inability to trace simple code	SM
misconceptions (about sequence, assignment, mental models and the notional machine)	SM
errors due to cognitive overload	SM/Preop
reluctance to trace	SM/Preop
ability to trace but not explain code	Preop
reliance on specific values	Preop

Table 6: Novice Programmer Behaviour

Next, we address each of the remaining questions:

Why do students get particular questions wrong?

There are a number of reasons, including guessing, misconceptions, inability to work with abstractions; and inability to focus on more than one element of a scenario.

Can a student have disparate levels of ability with two tasks which test similar programming concepts?

This behaviour was in fact evident with the tasks requiring students to trace code, then to reason about its purpose. A preoperational student can trace code, but they do not develop the ability to reason about its purpose until the concrete operational stage.

Why are some students unable to work with abstractions?

Ability to work with abstractions is not evident until the concrete operational stage. Based on our quantitative results , only the 12% of students who answered all the week 2 test questions correctly were likely to be reasoning at the concrete operational stage at that time, and only those students were manifesting concrete operational skills late in semester.

These results are consistent with our previous studies (Ahadi and Lister 2013; Ahadi et al. 2014) and means that most students are still manifesting sensorimotor and preoperational reasoning at the end of their first semester. Our think aloud studies support this. These results suggest that introductory programming educators are underestimating the foreignness to students of concepts taught very early in semester as well as their inability to reason abstractly.

10 Pedagogical Discussion

While it may be up to each student to practise and improve within a neo-Piagetian stage, we believe the teacher's role is to assist the students to transition from one neo-Piagetian stage to the next. We now offer suggestions on how they might facilitate that. As a general rule we agree with Bruner (1960):

It is into the language of (the novice's) internal structures that one must translate ideas if the (novice) is to grasp them.

10.1 From Sensorimotor to Preoperational

A sensorimotor student who guesses cannot be aware of *which* reasoning is accurate without external feedback. Until they have external feedback they are unlikely to resolve their misconceptions. Teachers should facilitate environments that encourage deliberate, supported practice (Guzdial 2014). We speculate that students who have not had external feedback "hedge their bets" in exams in the hope that one of the strategies is correct and will at least get them part marks.

Teachers should begin by offering students one-liner single-concept tasks. The earliest tasks should be purely literal expressions with gradual progression to univariate expressions. Teachers should be aware of and discourage rote learning and pattern matching, as that delays the transition to a higher stage.

Teach students how to trace code systematically, for example with a trace table, using appropriate values (test categories and cases). Furthermore, test them to ensure that they *are* tracing correctly.

Students at the sensorimotor stage require, more than anything else, that their misconceptions are corrected. For example: "what is an assignment statement?" or "what can (and can't) a variable do?". When students have overcome any misconceptions (especially about variables, assignment and sequence) and have a clear idea of the notional machine, and can start to trace code reliably, they are probably reasoning at the preoperational stage.

10.2 From Preoperational to Concrete

Teachers should gradually increase the complexity of the tasks with multivariate expressions and more complex code. Roles of variables (Kuittinen and Sajaniemi 2004) is one example of useful cognitive concepts that encourage abstract reasoning. In general, there should be a focus on tracing and explaining tasks with code writing tasks secondary.

10.2.1 Tracing and Explaining Code

Give preoperational students a complete function or very small program that does something interesting – perhaps with visual impact. Set them the task of experimenting with the code by making small, superficial changes. Give them practice at interpreting the results of a trace (i.e., identifying invariants and explaining the code's overall purpose). A good assessment task at this stage is to supply "buggy" code where the skills students have developed (above) are used to fix the code.

10.2.2 Abstract Tracing

Preoperational students are heavily reliant on specific values in variables to reason about code. This reliance diminishes as they become more proficient with programming and they develop an ability to trace "abstractly". In other words they are able to compute the effect of the code without using specific values. This ability to start working with abstractions signals the transition into concrete operational reasoning. Jim, for example, tried unsuccessfully to trace code abstractly (i.e., without specific values). However, he then succeeded by resorting to the use of specific values. He, and other preoperational students, will develop abstract tracing skills with persistent practice and challenges that require more mature strategies until they learn to reason about and work with abstractions. Tracing abstractly also means that the trace need not be complete in order to determine the code's purpose. A student transitioning into concrete operational stage may be able to short-circuit a trace because they can also simultaneously process a number of features of a block of code (e.g., in a loop). Only once students have begun to develop those sorts of reading skills will they begin to write code systematically.

11 References

Ahadi, A. and Lister, R. (2013): Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant? Proc. of *Ninth Annual International ACM Conference on International Computing Education Research (ICER 2013),* San Diego, CA. 123-128, ACM.

Ahadi, A., Lister, R. and Teague, D. (2014): Falling Behind Early and Staying Behind When Learning to Program. Proc. of *Psychology of Programming Interest Group (PPIG 2014),* Brighton, UK.

Biggs, J. B. and Collis, K. F. (1982): Origin and Description of the SOLO Taxonomy *Evaluating the quality of learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome).* New York: Academic Press Inc.

Boom, J. (2004): Commentary on: Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22,* 239-247.

Bruner, J. S. (1960): *The Process of Education*. London: Oxford University Press.

du Boulay, B. (1989): Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* 283-300. Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A. and Simon, H. A. (1993): *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.

Falkner, K., Vivian, R. and Falkner, N. J. G. (2013): Neo-Piagetian Forms of Reasoning in Software Development Process Construction. Proc. of *Learning and Teaching in Computing and Engineering (LaTiCE) 2013*, Macau. IEEE.

Feldman, D. H. (2004): Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 175-231.

Guzdial, M. (2014). Anyone Can Learn Programming: Teaching > Genetics. BLOG@CACM http://m.cacm .acm.org/blogs/blog-cacm/179347-anyone-can-learn-programming-teaching-genetics/fulltext 2014.

Kuittinen, M. and Sajaniemi, J. (2004). Teaching Roles of Variables in Elementary Programming Courses. ITiCSE '04. Leeds, UK, ACM.

Lister, R. (2011): Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. Proc. of *13th Australasian Computer Education Conference (ACE 2011)*, Perth, WA. **114:**9-18, ACS.

Lopez, M., Whalley, J., Robbins, P. and Lister, R. (2008): Relationships between Reading, Tracing and Writing Skills in Introductory Programming. Proc. of *ICER '08*, Sydney, Australia. ACM.

Siegler, R. S. (1996): *Emerging Minds*. Oxford: Oxford University Press.

Teague, D., Corney, M., Ahadi, A. and Lister, R. (2013): A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers. Proc. of *15th Australasian Computing Education Conference (ACE 2013)*, Adelaide, Australia. **136:**87-95, ACS.

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A. and Lister, R. (2012): Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming. Proc. of *Australasian Association for Engineering Education Conference (AAEE 2012)*, Melbourne.

Teague, D. and Lister, R. (2014a). Longitudinal Think Aloud Study of a Novice Programmer. *Australasian Computing Education Conference (ACE 2014).* Auckland, New Zealand, ACS. **148**.

Teague, D. and Lister, R. (2014b): Blinded by their Plight: Tracing and the Preoperational Programmer. Proc. of *Psychology of Programming Interest Group (PPIG) 2014*, Sussex, UK.

Teague, D. and Lister, R. (2014c): Manifestations of Preoperational Reasoning on Similar Programming Tasks. *Australasian Computing Education Conference (ACE 2014).* Auckland, New Zealand. **148**, ACS.

Appendix: The Test 1 Questions

Q1 In the boxes, write the values in the variables after the following code has been executed:

```
a = 1
b = 2
a = 3
```

The value in `a` is $\boxed{3}$ and the value in `b` is $\boxed{2}$

Q2 In the boxes, write the values in the variables after the following code has been executed:

```
r = 2
s = 4
r = s
```

The value in `r` is $\boxed{4}$ and the value in `s` is $\boxed{4}$

Q3 In the boxes, write the values in the variables after the following code has been executed:

```
p = 1
q = 8

q = p
p = q
```

The value in `p` is $\boxed{1}$ and the value in `q` is $\boxed{1}$

Q4 In the boxes, 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
```

The value in `x` is $\boxed{5}$ `y` is $\boxed{5}$ and `z` is $\boxed{5}$

Q5 In the boxes, write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 0

z = x
x = y
y = z
```

The value in `x` is $\boxed{5}$ `y` is $\boxed{7}$ and `z` is $\boxed{7}$

Q6 In Q5 above, what do you observe about the final values in x and y? Write your observation (in one sentence) in the box below.

Sample solution: The values in `x` and `y` were swapped.

Q7 The purpose of the following three lines of code is to swap the values in variables `a` and `b`, for any set of possible 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. Assume that variables i, j and k have been declared and initialised.

```
j = i
i = k
k = j
```

Sample solution: Swaps the values in `i` and `k`.

Q8 Assume the variables `first` and `second` have been initialised. Write code to swap the values stored in `first` and `second`.

Sample solution:

```
temp   = first
first  = second
second = temp
```

Dynamic evaluation trees for novice C programmers

Matthew Heinsen Egan[1] **Chris McDonald**[2]

School of Computer Science and Software Engineering
University of Western Australia
Crawley, Western Australia 6009
Email: [1]m.heinsen.egan@graduate.uwa.edu.au
[2]chris.mcdonald@uwa.edu.au

Abstract

The *dynamic evaluation tree* is a method of visualizing expression evaluation that annotates a program's source code with expression results. It is intended to reduce students' visual attention problems by removing the need to alternate between disparate source code and expression evaluation windows. We generalise the dynamic evaluation tree to support arbitrary expressions in the C programming language, and present the first ever implementation for a novice-focused program visualization and debugging tool.

Keywords: Novice programmers, debuggers, software visualization

1 Introduction

Expression evaluation can be difficult for novice programmers to comprehend. An incomplete understanding of expression evaluation may make it exceedingly difficult for novices to identify and correct malformed expressions in their own code. In a multi-institutional study of novice debuggers, Fitzgerald et al. (2008) found that the most difficult bugs for their subjects to find and fix were arithmetic bugs (in particular) and malformed statement bugs (in general). Effective visualization of expression evaluation may assist novice programmers to construct knowledge of expression evaluation, including the behaviour of individual operators, and to debug programs containing malformed expressions.

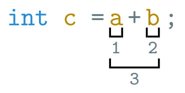
Brusilovsky & Spring (2004) discussed a tutoring system designed to assist students learn expression evaluation in the C programming language, stating:

> "For the students in our programming and data structure courses based on C language, expression evaluation is one of the most difficult concepts to understand. They have problems with both understanding the order of operator execution in a C expression and understanding the semantics of operators."

The web-based system, WADEIn, visualizes the step-by-step evaluation of expressions consisting of mathematical and logical operators with `int` and `double` type variables. More than 80% of students felt that the system helped them to understand C operations.

Many existing software visualization systems use a dedicated "*expression evaluation*" area to visualize the individual operations performed during an expression's evaluation (e.g. Jeliot 3, as presented by Moreno et al. (2004); and The Teaching Machine described by Bruce-Lockhart et al. (2007)). Animation is commonly used to relate operations to the expression's source code, and operands to memory visualizations. For example, if an evaluated operator's operand is a variable, then the variable's value might "fly in" from the memory visualization.

Lahtinen & Ahoniemi (2009) introduced the "*dynamic evaluation tree*" for visualizing expression evaluation by annotating above or below a program's source code, e.g.:



This concept was primarily motivated by the results of an eye-tracking study of Jeliot 3 users, which found that novice programmers "*either switch their visual attention repeatedly between different windows or concentrate all the time on one of the windows*" (Lahtinen & Ahoniemi 2009). The dynamic evaluation tree is intended to integrate expression evaluation and source code representation, thus reducing the switching of visual attention required by novice programmers. Lahtinen and Ahoniemi discussed the potential of adding the dynamic evaluation tree to the VIP C++ program visualization system, but unfortunately this work has not been continued.

Annotations in a dynamic evaluation tree maintain a visual relationship to their associated source code, as opposed to animated visualizations which only briefly show this relationship (e.g. by having the relevant source code "fly in" to the evaluation area). This explicit visualization of the expression evaluation's history may reduce the need for students to step backwards and forwards, and clarify the relationships between individual operations.

This paper discusses our implementation of a dynamic evaluation tree for the novice-focused program visualization and debugging tool *SeeC*. Section 2 generalises the dynamic evaluation tree to support arbitrary expressions in the C programming language. Section 3 describes our implementation. Section 4 discusses integration with SeeC's other components. Section 5 compares our implementation with traditional visualizations of expression evaluation. Section 6 discusses limitations in our implementation and identifies future work. Finally, Section 7 summarizes our discussion.

2 General C programs

Despite its age, the C programming language still holds an important place in computing education. While few traditional Computer Science courses teach C as an introductory programming language in their foundation years, C remains important in the later teaching of operating systems and computer networking. C still enables students to understand the close relationship between programming languages and hardware in increasingly important subjects such as robotics, embedded systems, and wearable computing, and these subjects are often required by students other than future computer scientists. Novice C programmers are not necessarily novice programmers, and those whose entire exposure to programming has been through safe languages still have to address many challenges. Many of the traditional problems with C, such as its practice of leaving much as "defined to be undefined" and the challenges of writing portable code across disparate operating systems and architectures, have been addressed by detailed official standards, shifting the pressure to those teaching C to do so well.

The simplicity and familiarity of the dynamic evaluation tree is a great strength. It provides a concise, clear representation of complex expression evaluations. Implementing the dynamic evaluation tree for SeeC required us to support arbitrary expressions in the C programming language, introducing several complicating details. This section discusses these complications and the approaches that we employed to ensure that the dynamic evaluation tree retains its conciseness, clarity, and, we believe, usefulness.

The simplest problem is that an annotation's text may be wider than the annotated expression's source code. This may obscure the visual relationship between the annotation and source code, and could lead to overlapping annotations. We prevent this simply by truncating annotation text to the width of the expression's source code. Students can view the complete text by hovering the cursor over the annotation.

The dynamic evaluation tree is designed to annotate a single line of source code, but students are free to write an expression over multiple lines. This may be uncommon in novice programmers' code, but our general purpose implementation must account for it. Our straightforward solution is to reformat the expression's source code, displaying it on a single line while the dynamic evaluation tree is active.

The C programming language's *preprocessor* may also necessitate the use of modified source code to represent expressions, as a single macro may expand to multiple sub-expressions. If each expression had at most a single child, we could simply stack the annotations. For example, consider a typical implementation of the NULL macro:

```
#define NULL ((void*)0)
```

$$\underbrace{\underbrace{\texttt{NULL}}_{\text{integer literal: } \texttt{0}}}_{\text{cast: } \texttt{0x0}}$$

For more complex macros the visualization will become increasingly crowded. As an example, consider the sys/stat.h header's S_ISREG macro, defined by The Open Group Base Specifications Issue 7 thus[1]: *"The value m supplied to the macros is the value of st_mode from a stat structure. The macro shall evaluate to a non-zero value if the test is true; 0 if the test is false."* A typical implementation of this macro is:

```
#define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```

[1]http://pubs.opengroup.org/onlinepubs/9699919799/

Visualizing the complete tree created by using the macro S_ISREG would expose students to unnecessary, potentially confusing implementation details. Thus it may be best to employ a black box representation by restricting the visualization to the "input" and "output" nodes: in this case, m and the result of the == operator, respectively. Conversely, it should be possible for students to observe the behaviour of code produced by their own macros: showing the preprocessed code will allow students to observe their macro's expansion, and a dynamic evaluation tree visualizing the resulting expression's behaviour.

In the C programming language an expression may designate an object; such expressions are termed *lvalues*[2]. For example, in line 4 of Listing 1 the expressions total, iptr, and iptr[i] are lvalues. An expression which does not designate an object, for example the expression total + iptr[i], is commonly referred to as an *rvalue*[3].

Listing 1 Summing an array of int values

```
1 int sum_ints(const int *iptr, size_t n) {
2   int total = 0;
3   for (size_t i = 0; i < n; ++i)
4     total = total + iptr[i];
5   return total;
6 }
```

Some expressions require an lvalue, e.g. the unary & operator produces the address of the designated object, and the ++ operator increments the value stored in the designated object. For most other uses an lvalue is converted to the value stored in the designated object, e.g. iptr[i] in Listing 1. In terms of the language implementation we might consider this to represent the value being loaded from memory. The behaviour of such lvalues poses a question for the visualization of dynamic evaluation trees: should we show the designated object, the value that was stored in the designated object when the expression was evaluated, or both? An explicit relationship to the designated object will allow students to see where values are coming from. This may be particularly useful for array accesses and pointer dereferences. However, showing the value stored in the designated object may be confusing if the value changes after the expression is evaluated, for example:

$$\texttt{number} = \underbrace{\texttt{10} / \underbrace{\texttt{number}}_{2}}_{5};$$

When this assignment expression is completed the value 5 will be stored in the object designated by number. However, the value of the number expression on the right hand side should still be 2, otherwise the division's result is nonsensical. Our approach is to show two nodes: one for the lvalue, and one for the rvalue it was converted to during evaluation. The lvalue is annotated with descriptive placeholder text rather than the designated object's value. When the student moves the cursor over this annotation, the designated object is highlighted in SeeC's standard memory visualization.

Expressions with struct or union types are difficult to represent within an annotation, as they may contain numerous fields and values, thus causing the

[2]ISO/IEC 9899:2011 (The C11 Standard) §6.3.2.1.1

[3]ISO/IEC 9899:2011 uses the term "value of an expression".

textual representation to be far larger than the expression's source code. If the expression is an lvalue then we again show a placeholder and direct students to a memory visualization for the complete value. This is not possible for rvalue expressions, so we truncate the annotation when necessary and show the complete value when the student hovers the mouse cursor over the node.

Pointers, often described as a threshold concept in Computer Science (Boustedt et al. 2007, Rountree & Rountree 2009), are a source of great difficulty for novice C programmers, and so it is essential to effectively visualize pointer type expressions. The raw value of a pointer is generally not important for novice C programmers, rather they are concerned with whether pointers are valid and which objects they reference. Displaying the value of the referenced object could visualize this information, but might cause dangerous misconceptions about the semantics of pointers. We handle this similarly to lvalues: the node's annotation contains placeholder text, and when students move the mouse cursor over the node the referenced object is highlighted in SeeC's memory visualization. The placeholder text indicates whether the pointer is valid, invalid, opaque, or `NULL`.

3 Implementation

We implemented a dynamic evaluation tree as an extension to the SeeC project: a system for novice C programmers that performs execution tracing with automatic runtime error detection, and provides program visualization of the recorded execution traces, as described by Heinsen Egan & McDonald (2014). SeeC itself is built upon the Clang project[4]: a modular collection of libraries which implement a front-end for compiling C, C++, Objective C, and Objective C++, but are also designed to support diverse uses by external clients. Students reviewing an execution trace can step forwards or backwards to any point in the process' execution. The SeeC system provides a "*recreated state*" of the process, which we use to generate the dynamic evaluation tree.

The "*recreated state*" of the function that was executing provides us with the "*currently active*" statement, which is either partially evaluated or has just completed evaluation (in which case it may have produced a value). If this statement is an expression then we walk up Clang's Abstract Syntax Tree to find the "*top-level*" expression, i.e. the first node whose parent is not also an expression. The top-level expression is the root of our dynamic evaluation tree, ensuring visualizations remain consistent during the evaluation of complex expressions.

We produce a modified representation of the expression's source code using Clang's lexing and preprocessing systems. We iterate over each preprocessed token in the expression's source code. If the token was expanded from a user-defined macro then we add all of the expanded tokens to the modified representation. If the token was expanded from a macro defined in a system header, then we add the raw tokens covering the range the macro was expanded from. If the token was not expanded from a macro then we simply add it as-is. Tokens do not include newlines, so this method also fulfils our requirement of producing a single line of source code.

For an example of handling user-defined macros, consider Listing 2 (above right). The top-level expression is the initializer of `metres`: from the `2` to the final closing parenthesis. The tokens `2` and `*` are added to the modified representation as-is, because they do not involve macro expansion. The next token, `6372797`, is expanded from a macro defined in the user's source code, so we add the expanded tokens to the modified representation. All remaining tokens are added as-is, because they do not involve macro expansion.

[4]http://clang.llvm.org

Listing 2 User defined macro

```
#define EARTH_RADIUS_IN_METRES 6372797

double metres   = 2 * EARTH_RADIUS_IN_METRES
                * asin(sqrt(x));
```

For an example of handling macros that are defined in system headers, consider the use of `S_ISREG` shown in Listing 3 (below). The top-level expression is the `if` statement's condition. The first token is expanded from a macro that was defined in a system header, so we find the area that the macro was expanded from and add the *raw* tokens to the modified representation: `S_ISREG(st.st_mode)`. The expanded tokens are discarded.

Listing 3 System macro expansion

Raw:
```
if (S_ISREG(st.st_mode)) {
```
Preprocessed:
```
if (((((st.st_mode)) & 0170000) == (0100000))) {
```

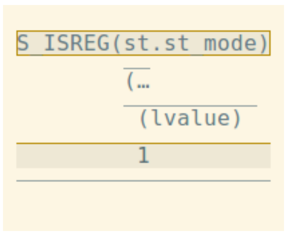


Figure 1: System macro evaluation

We annotate only the topmost expression from the *body* of expanded system macros in order to produce the "black box" representation discussed in Section 2. For example, consider the dynamic evaluation tree for Listing 3 shown in Figure 1 (above): the topmost node from the expanded body is shown (the `==` operator, with value `1`), and all other nodes from the expanded body are hidden (e.g. the `&` operator). We display nodes represented by the expanded *argument* to visualize the behaviour of the student's code.

The system next determines each expression's annotation text. SeeC provides information about the value produced by any expression's most recent evaluation. For example we will refer to the nodes in Figure 1. If the node's expression is a pointer or an lvalue then we use descriptive placeholder text for the annotation (e.g. the "`(lvalue)`"). For all other expressions we use SeeC's string representation of the value (e.g. the "`1`"). Annotation text that is too wide for the node is truncated, e.g. the node representing `st` is truncated from the full text "`(lvalue)`".

SeeC automatically detects several kinds of runtime errors during program execution, and provides information about detected errors during replay. We draw a dotted red line surrounding a statement's node if a runtime error was detected during that statement's execution, so that students may quickly locate errors in the dynamic evaluation tree. Figure 2 shows the dynamic evaluation tree rendered when an invalid index is used as a subscript of argv.

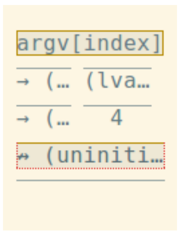


Figure 2: Statement with detected runtime error

The dynamic evaluation tree is a concise visualization of expression evaluation, but more information is available. To maintain clarity we use the "drill down" design, showing the following details in a tooltip when the mouse cursor hovers over an annotation:

- The complete annotation text.
- The expression's type. This allows students to observe the behaviour of type conversions (both implicit and explicit), and may be useful for debugging arithmetic errors (e.g. accidental use of integer division).
- A natural language explanation of the expression, as described by Heinsen Egan & McDonald (2014).
- A natural language description of any runtime errors that SeeC detected during the statement's execution.

Figure 3 shows an example of this tooltip. Further information and functionality is provided by integrating with, and deferring to, SeeC's other systems.

4 Integration with SeeC

The SeeC tool shows several complementary visualizations when replaying execution traces. We often reference these visualizations because the dynamic evaluation tree alone cannot conveniently represent all expression values, as we discussed in Section 2. In several situations we use placeholder text and direct students to other visualizations, e.g. to view an lvalue's designated object in memory.

Moving the cursor over a node in the dynamic evaluation tree causes its associated expression to be highlighted, in both the modified representation of the source code and the regular source code window. If the expression is an lvalue and has been evaluated, then its designated object will also be highlighted in the memory visualization window. Figure 3 shows both highlights: lon2 is outlined in the source code window on the left, and lon2's designated object is highlighted in the memory visualization on the right. If the expression is a pointer then the pointee object is also highlighted; this is necessary for observing rvalue pointers.

SeeC provides "*contextual navigation*" options, which we have also made accessible through the dynamic evaluation tree. Right clicking on any node provides navigation options based on the associated expression: move backward to the last time the expression was evaluated, or move forward to the next time the expression was evaluated. For lvalue expressions we also provide navigation based on the designated object's memory: move backward to its allocation, move forward to its deallocation, move backward to the prior time the memory was modified, or move forward to the next time the memory was modified.

5 Comparing visualizations

Our dynamic evaluation tree is not yet integrated with SeeC's source code window in the manner proposed by Lahtinen & Ahoniemi (2009): it occupies its own window within SeeC, in the manner of traditional expression evaluation visualizations. In this section we compare our implementation with existing visualizations, arguing that it offers several benefits despite not yet consolidating these windows. We will compare these visualizations with reference to Cognitive Load Theory as described by Sweller et al. (1998), and to the guidelines that Ware (2008) provides for information visualization based on current understandings of human perception and cognition.

Cognitive Load Theory provides guidelines for representing information to optimize intellectual performance and promote knowledge acquisition. These guidelines relate to optimizing the use of working memory: information must be in working memory in order to be processed, and working memory is extremely limited. Effective representations decrease *extraneous cognitive load*: the effect on working memory load of the manner in which information is presented, or of the activities required by students, i.e. that which is not intrinsic to the material at hand. Decreasing extraneous cognitive load enables students to devote more working memory to performing tasks and acquiring knowledge. This is particularly important when dealing with material that has a high *intrinsic cognitive load*. The *Split-Attention Effect* described by Sweller et al. (1998) is especially relevant to our comparison of visualizations. The Split-Attention Effect occurs when a student must mentally integrate two distinct sources of information in order to understand them, e.g. textual information that refers to a diagram, where neither the textual information nor the diagram are effective independently.

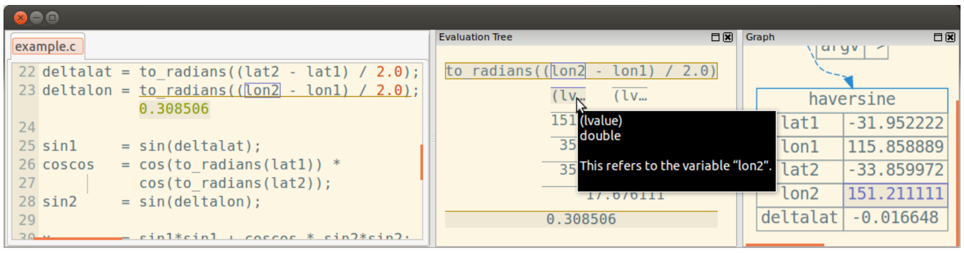
> On the basis of dozens of experiments under a wide variety of conditions, the evidence suggests overwhelmingly that it has negative consequences and should be eliminated wherever possible. (Sweller et al. 1998)

Ware (2008) provides a wealth of information concerning the effective design of information visualizations. Of particular relevance to program visualization systems are the recommendations on *optimizing the cognitive process*:

> The ideal cognitive loop involving a computer is to have it give you exactly the information you need when you need it. This means having only the most relevant information on screen at a given instant. It also means minimizing the cost of getting more information that is related to something already discovered. This is sometimes called *drilling down*. (Ware 2008)

There are two possibilities when attempting to get information related to something already discovered: either it is displayed somewhere else on the screen, or the user must perform some action to cause it to be displayed. Eye movements are much faster than mouse movements, but displaying too much information on screen will increase the difficulty of searching for any particular piece of information.

With this information in hand, let us now compare SeeC's implementation of dynamic evaluation trees



```
deltalat = to_radians((lat2 - lat1) / 2.0);
deltalon = to_radians((lon2 - lon1) / 2.0);
           0.308506

sin1     = sin(deltalat);
coscos   = cos(to_radians(lat1)) *
           cos(to_radians(lat2));
sin2     = sin(deltalon);
```

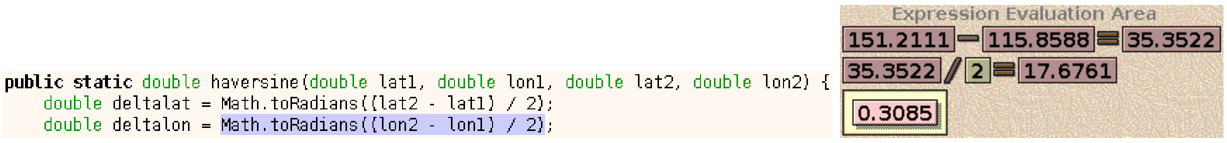
```
to_radians((lon2 - lon1) / 2.0)
(lv…   (lv…
151
35
35
17.676111
0.308506
```

(lvalue)
double

This refers to the variable “lon2”.

haversine	
lat1	-31.952222
lon1	115.858889
lat2	-33.859972
lon2	151.211111
deltalat	-0.016648

Figure 3: SeeC’s highlighting and tooltip



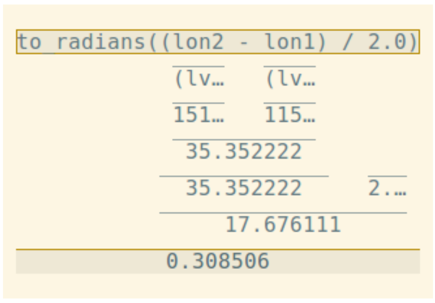
```
public static double haversine(double lat1, double lon1, double lat2, double lon2) {
    double deltalat = Math.toRadians((lat2 - lat1) / 2);
    double deltalon = Math.toRadians((lon2 - lon1) / 2);
```

```
151.2111 − 115.8588 = 35.3522
35.3522 / 2 = 17.6761
0.3085
```

Figure 4: Jeliot 3 source code (left) and expression evaluation (right)

with the existing visualizations of expression evaluation used by novice focused programming tools.

Figure 4 shows a completed expression evaluation in Jeliot 3: operators and values are shown in the expression evaluation area, but students must consult the source code window for any other information about the expression. Thus the observed repeated switching of visual attention that motivated Lahtinen & Ahoniemi (2009) to propose the dynamic evaluation tree. This is a clear example of the Split-Attention Effect: the expression evaluation area alone is unintelligible, and students are forced to mentally integrate information from other windows in order to make sense of it. SeeC’s dynamic evaluation tree, shown in Figure 5, contains a modified representation of the top-level expression’s source code, so switching visual attention to the main source code window is only necessary when referring to other expressions or to the original representation.



```
to_radians((lon2 - lon1) / 2.0)
       (lv…     (lv…
       151…     115…
         35.352222
         35.352222      2.…
              17.676111
      0.308506
```

Figure 5: SeeC

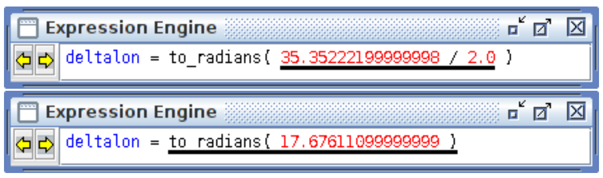
The dynamic evaluation tree maintains a clear mapping between values and source code: the expression that produced a value occupies the same horizontal space as the value’s node. Consider finding the expression that produced the value `35.3522` used in the division operation: in Jeliot 3 students must find the corresponding division operator in the source code window and then identify the left operand; in SeeC students can simply look at the top of the dynamic evaluation tree to see the source code occupying the same space as the value, or move their mouse cursor over the value to have that source code automatically highlighted. If a student wishes to determine *why* this expression produced this value using Jeliot 3 then they must find the correct subtraction operation in the evaluation history, perhaps by searching the right-hand side of the operations for the chosen value. Students using SeeC can simply look at the value’s children in the dynamic evaluation tree.

SeeC consistently uses highlighting to visualize relationships and thus minimize the cost of finding related information, both within the dynamic evaluation tree and between different visualizations. We can see this highlighting in Figure 3 (above). The active expression is outlined in yellow in both the source code window and the dynamic evaluation tree. The annotation under the mouse cursor has its associated expression highlighted in violet, and as it is an *lvalue* the designated object is similarly highlighted in the memory visualization. This method is applied consistently throughout SeeC, e.g. moving the mouse cursor over an expression in the source code window will highlight the corresponding expression (and its produced value) in the dynamic evaluation tree.

In Jeliot 3, when a variable’s value is used in an expression an animation shows the value “flying in” to the expression evaluation area. This provides only a transient association which, if it is important to the student’s task, must be held in working memory, unnecessarily burdening their working memory load. Furthermore, the student may not know whether the association is important at the time the animation occurs, and there is no option to display the association after the fact: instead, students must determine the association themselves by mentally integrating information from Jeliot 3’s multiple displays.

Bruce-Lockhart et al. (2007) described The Teaching Machine, a program visualization system supporting subsets of the Java and C++ languages, which also uses highlighting to illustrate relationships between different visualizations. Figure 7 provides an example: the active top-level expression’s source code is highlighted in yellow, and the active sub-expression is an *lvalue* whose designated object (`lon1`) is also highlighted in yellow. If the student wishes to see the relationship between a different sub-expression and the values in memory, they must step backwards or forwards until that sub-expression is active.

The Teaching Machine visualizes expression evaluation using *expression rewriting*, in which an evaluated sub-expression’s source code is replaced with its resulting value. Figure 6 shows the rewrite caused by an evaluation in The Teaching Machine: the underlined source code is the active sub-expression, which will be replaced by its result when the student steps forward. This visualization shows no history: students must step backwards to see previous operations. Furthermore, an operation’s operands and result are not simultaneously visible, so considering an operation requires a student to hold relevant information in working memory while stepping forwards or backwards. Effectively, the student is required to mentally integrate information from two visualizations which cannot be displayed simultaneously. The dynamic evaluation tree does not require this information to be held in working memory, because it is always accessible via rapid eye movements or mouse hovering.



```
deltalon = to_radians( 35.35222199999998 / 2.0 )
```

```
deltalon = to_radians( 17.67611099999999 )
```

Figure 6: The Teaching Machine 2's rewriting

Expression rewriting is also used by WADEIn, a web-based tool designed to help students construct knowledge of C's expression evaluation rules, presented by Brusilovsky & Spring (2004). WADEIn annotates the source code of an expression with the order in which the individual sub-expressions will be evaluated (shown as numbers beneath the sub-expressions). The evaluation of the complete expression is visualized by a "shrinking copy" of the source code: the active sub-expression is copied into an "evaluation area", its evaluation is visualized, and the result then replaces the original sub-expression in the "shrinking copy". Only the active sub-expression's evaluation is shown, so students must step backwards and forwards to observe the evaluation of different sub-expressions. WADEIn is a tutoring system for isolated expressions: it supports only mathematical and logical operators with `int` and `double` type variables. The system tracks the student's exposure to different operators, increasing the speed of animation and removing certain sub-steps as the student's "level of knowledge" increases.

The dynamic evaluation tree is the only method of visualization that shows every step of a complex expression's evaluation in a single image while maintaining relationships from evaluated sub-expressions to their original source code, and from lvalue expressions to their designated objects. Considering the advice and information provided by Sweller et al. (1998) and by Ware (2008), we believe the dynamic evaluation tree is a significant advancement in terms of both reducing extraneous cognitive load and optimizing the process of finding information that is related to something already discovered.

6 Limitations and future work

Future developments should be guided by the requirements of novice programmers learning the C programming language, thus the most important remaining task is to evaluate the dynamic evaluation tree's usage by novice programmers. We are currently investigating SeeC's usage by students in our second year course covering the C programming language and Operating Systems, and will be collecting feedback from students including their perceptions of the dynamic evaluation tree visualization and their suggestions for future development. During our own development and use of the dynamic evaluation tree we have identified some potential areas of investigation, which we describe in the remainder of this section.

We use Clang's expressions to generate our dynamic evaluation tree. This reduces our system's implementation requirements and provides robust, complete support for the C programming language, but could expose technical details that may confuse novice programmers. We hide some information from students: for example, in Figure 5, we hide the expressions representing the reference to `to_radians` and its decay to a function pointer. It may be useful to provide an option to display all expressions, or to implement an adaptive system that reveals technical details when a student's knowledge is sufficiently advanced.

User-controlled information eliding may also be useful for handling macro expansion. Our implementation either fully expands or does not expand macros, but in some situations it may be useful to show a *partial expansion*. Listing 4 shows a definition for the function-like macro `S_ISREG`; a raw use of this macro; and a partial expansion of this use, in which the expanded tokens have not undergone *rescanning* which would have expanded `S_IFMT` and `S_IFREG`. Showing `S_IFMT` and `S_IFREG` rather than their expanded numeric constants may be more informative than the fully preprocessed code (e.g. shown in Listing 3). Students could interactively control whether individual macros are expanded, allowing them to inspect the preprocessor's actions and to select an appropriate representation for the task at hand.

Listing 4 Partial macro expansion

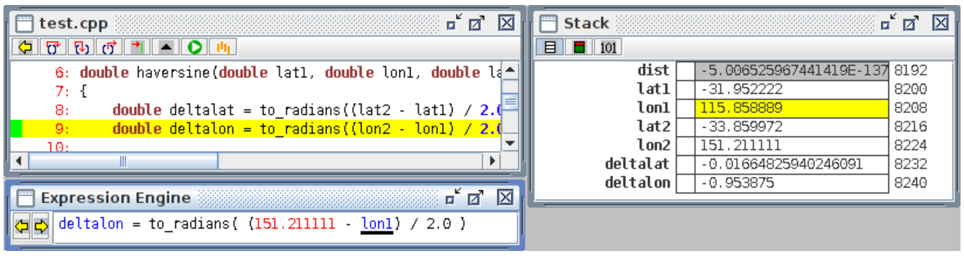
Macro definition:
```
S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```
Raw:
```
S_ISREG(st.st_mode)
```
Partially expanded:
```
(((st.st_mode) & S_IFMT) == S_IFREG)
```

The dynamic evaluation tree visualizes the values produced by each expression, but it does not represent expressions' side effects. For example, a postfix increment operator's node would show the value loaded from the operand's designated object, but would not indicate that the object's value was modified. This problem is generalised by annotating function calls, which may have numerous side effects. It may be useful to visually indicate that an annotation's associated expression caused some side effects. The exact nature of the side effects could be represented in the tooltip produced by hovering the mouse cursor over the annotation.

7 Summary

The dynamic evaluation tree concisely visualizes expression evaluation while maintaining a visual relationship between each expression's source code and its produced value. The complete history of a complex expression evaluation can be shown in a single static frame, enabling students to rapidly scan each step of the evaluation. In this paper we generalised the dynamic evaluation tree to account for arbitrary expressions in the C programming language, presented our implementation of the dynamic evaluation tree for the novice-focused program visualization and debugging tool SeeC, and compared this implementation to previous visualizations of expression evaluation.

We believe that the complicating factors discussed and mitigated within this work will support attempts to implement the dynamic evaluation tree in other novice-focused tools, regardless of their supported programming languages. For example, the difficulties of representing pointers may also apply to the representation of references in Java or Python.



```
double haversine(double lat1, double lon1, double la
{
    double deltalat = to_radians((lat2 - lat1) / 2.
    double deltalon = to_radians((lon2 - lon1) / 2.
```

```
     dist  -5.006525967441419E-137  8192
     lat1  -31.952222               8200
     lon1  115.858889               8208
     lat2  -33.859972               8216
     lon2  151.211111               8224
 deltalat  -0.01664825940246091     8232
 deltalon  -0.953875                8240
```

```
deltalon = to_radians( (151.211111 - lon1) / 2.0 )
```

Figure 7: The Teaching Machine 2's highlighting

The dynamic evaluation tree was introduced by Lahtinen & Ahoniemi (2009) with the intention of reducing novice programmers' switching of visual attention while using program visualization tools. To our knowledge, we have presented the first implementation of this concept. We believe this is a robust, maintainable implementation and yet its development was straightforward, which speaks to the underlying SeeC system's potential as a foundation for novice-focused program visualization research.

Finally, this implementation enables investigation of the dynamic evaluation tree's usefulness for novice programmers learning the C programming language.

8 Acknowledgements

This research is partially supported by an Australian Postgraduate Award.

References

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K. & Zander, C. (2007), 'Threshold concepts in computer science: Do they exist and are they useful?', *SIGCSE Bull.* **39**(1), 504–508.

Bruce-Lockhart, M., Norvell, T. S. & Cotronis, Y. (2007), 'Program and algorithm visualization in engineering and physics', *Electron. Notes Theor. Comput. Sci.* **178**, 111–119.

Brusilovsky, P. & Spring, M. (2004), Adaptive, Engaging, and Explanatory Visualization in a C Programming Course, *in* 'ED-MEDIA'2004 - World Conference on Educational Multimedia, Hypermedia and Telecommunications', pp. 21–26.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008), 'Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers', *Computer Science Education* **18**(2), 93–116.

Heinsen Egan, M. & McDonald, C. (2014), Program visualization and explanation for novice C programmers, *in* 'Sixteenth Australasian Computing Education Conference (ACE 2014)', Vol. 148 of *CRPIT*, ACS, Auckland, New Zealand, pp. 51–57.

Lahtinen, E. & Ahoniemi, T. (2009), 'Dynamic evaluation tree for presenting expression evaluations visually', *Electronic Notes in Theoretical Computer Science* **224**, 41 – 46. Proceedings of the Fifth Program Visualization Workshop (PVW 2008).

Moreno, A., Myller, N., Sutinen, E. & Ben-Ari, M. (2004), Visualizing programs with Jeliot 3, *in* 'AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces', ACM, New York, NY, USA, pp. 373–376.

Rountree, J. & Rountree, N. (2009), Issues regarding threshold concepts in computer science, *in* 'Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95', ACE '09, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 139–146.

Sweller, J., van Merrienboer, J. & Paas, F. (1998), 'Cognitive architecture and instructional design', *Educational Psychology Review* **10**(3), 251–296.

Ware, C. (2008), *Visual Thinking for Design*, Morgan Kaufmann Publishers, 30 Corporate Drive, Suite 400, Burlington, MA, USA.

Author Index