Computing Education 2014

Proceedings of the
Sixteenth Australasian Computing Education Conference
(ACE 2014), Auckland, New Zealand,
20 – 23 January 2014

Jacqueline Whalley and Daryl D’Souza, Eds.

Volume 148 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.

Conferences in Research and Practice in Information Technology, Volume 148.

Copyright ©2014, Australian Computer Society. Reproduction for academic, not-for-profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:

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

Daryl D’Souza
School of Computer Science and Information Technology
RMIT University
Melbourne, VIC 3001
Australia
Email: daryl.d souza@rmit.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 148.
ISSN 1445-1336.

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

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 ................................................................. vii
Programme Committee ........................................... viii
Organising Committee ............................................ ix
Welcome from the Organising Committee ......................... x
CORE - Computing Research & Education ........................ xi
ACSW Conferences and the Australian Computer Science Communications ....................................................... xiii
ACSW and ACE 2014 Sponsors ....................................... xv

## Contributed Papers

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Australian Digital Technologies Curriculum: Challenge and Opportunity</td>
<td>3</td>
</tr>
<tr>
<td><em>Katrina Falkner, Rebecca Vivian and Nickolas Falkner</em></td>
<td></td>
</tr>
<tr>
<td>Visualising Career Progression for ICT Professionals and the Implications for ICT Curriculum Design in Higher Education</td>
<td>13</td>
</tr>
<tr>
<td><em>Brian von Konsky, Charlynn Miller and Ashely Jones</em></td>
<td></td>
</tr>
<tr>
<td>ICT Curriculum and Course Structure: the Great Balancing Act</td>
<td>21</td>
</tr>
<tr>
<td><em>Nicole Herbert, Kristy de Salas, Ian Lewis, Julian Dermoudy and Leonie Ellis</em></td>
<td></td>
</tr>
<tr>
<td>Student and Staff Expectations of NZQA Levels</td>
<td>31</td>
</tr>
<tr>
<td><em>Mike Lopez and Dobrila Lopez</em></td>
<td></td>
</tr>
<tr>
<td>Longitudinal Think Aloud Study of a Novice Programmer</td>
<td>41</td>
</tr>
<tr>
<td><em>Donna Teague and Raymond Lister</em></td>
<td></td>
</tr>
<tr>
<td>Program Visualization and Explanation for Novice C Programmers</td>
<td>51</td>
</tr>
<tr>
<td><em>Matthew Heinsen Egan and Chris McDonald</em></td>
<td></td>
</tr>
<tr>
<td>Worked Examples in Computer Science</td>
<td>59</td>
</tr>
<tr>
<td><em>Ben Skudder and Andrew Luxton-Reilly</em></td>
<td></td>
</tr>
<tr>
<td>Manifestations of Preoperational Reasoning on Similar Programming Tasks</td>
<td>65</td>
</tr>
<tr>
<td><em>Donna Teague and Raymond Lister</em></td>
<td></td>
</tr>
<tr>
<td>Teaching Mobile Apps for Windows Devices Using TouchDevelop</td>
<td>75</td>
</tr>
<tr>
<td><em>Simon and David Cornforth</em></td>
<td></td>
</tr>
<tr>
<td>A Web-based Firewall Simulator Tool for Information Security Education</td>
<td>83</td>
</tr>
<tr>
<td><em>Zouheir Trabelsi and Umniya Mustafa</em></td>
<td></td>
</tr>
<tr>
<td>A Multi-tier Client-Server Project Employing Mobile Clients</td>
<td>91</td>
</tr>
<tr>
<td><em>Chris MacDonald</em></td>
<td></td>
</tr>
<tr>
<td>Teaching Model-Driven Software Development: Revealing the “Great Miracle” of Code Generation to Students</td>
<td>97</td>
</tr>
<tr>
<td><em>Andreas Schmidt, Daniel Kimmig, Klaus Bittner and Markus Dickerhof</em></td>
<td></td>
</tr>
</tbody>
</table>
How Difficult are Novice Code Writing Tasks? A Software Metrics Approach  ......................... 105
  Jacqueline Whalley and Nadia Kasto

Benchmarking a Set of Exam Questions for Introductory Programming  ......................... 113
  Judy Sheard, Simon, Julian Dermoudy, Daryl D’Souza, Minjie Hu and Dale Parsons

Developing A Framework to Assess Students Contributions during Wiki Construction  ........ 123
  Iwan Handoyo Putro, Angela Carbone and Judy Sheard

Solution Spaces  ................................................................. 133
  Nadia Kasto, Jacqueline Whalley, Anne Philpott and David Whalley

Introductory Programming Courses in Australia and New Zealand in 2013 - Trends and Reasons ... 139
  Raina Mason and Graham Cooper

Unblocking the Pipeline by Providing a Compelling Computing Experience in Secondary Schools:
  Are the Teachers Ready? ......................................................... 149
  Catherine Lang, Annemieke Craig and Gail Casey

Its Not Them, Its Us! Why Computer Science Fails to Impress Many First Years  ................ 159
  Rashina Hoda and Peter Andreae

Author Index  ................................................................. 163
Preface

Welcome to the Sixteenth Australasian Computing Education Conference (ACE2014). This year the ACE2014 conference, which is part of the Australasian Computer Science Week, is being held at AUT University, Auckland, New Zealand from 20 to 23 January, 2014.

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 19 full papers from the 40 papers submitted, giving an acceptance rate of 47%. The number of submissions was similar to the 39 papers submitted in the previous year. We again see a strong national and international presence, with submissions from Australia, Finland, New Zealand, Sweden United Arab Emirates, and the United States of America. A variety of topics are presented in this year’s papers, including: novice programming; assessment; curricula design; the first year experience; computing in schools and technologies for computer education. Many of the papers present new innovations and demonstrate high quality research.

This year we invited Professor Anthony Robins, from the Department of Computer Science, Auckland University, New Zealand to deliver an ACSW key address.

The Software Engineering Laboratory (SERL) at AUT University covered the ACE registration fee for ten PhD students to discuss and explore their research interests and career objectives with a panel of established researchers in computing education research. The doctoral consortium is chaired by Associate Professor Katrina Falkner from the University of Adelaide in Australia.

As with past ACE conferences, we are continuing to hold workshops. This year three workshops have been organized, these include: Catalyzing & Sustaining Change in Computing Education led by Professor Lynn Andrea Stein supported by a NSF grant; Computing A 21st Century Literacy led by Michael E. Caspersen, Palle Nowack and Tim Bell; Benchmarking exams questions for an introductory programming course led by Simon and Judy Sheard.

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:

* Benchmarking a set of exam questions for introductory programming
  Judy Sheard, Simon, Julian Dermoudy, Daryl D’Souza, Minjie Hu and Dale Parsons

One other paper was also highly commended:

* The Australian Digital Technologies Curriculum: Challenge and Opportunity
  Katrina Falkner, Rebecca Vivian and Nickolas Falkner

The best student paper was awarded to:

* Manifestations of Preoperational Reasoning on Similar Programming Tasks
  Donna Teague and Raymond Lister

We are grateful to SIGCSE for sponsoring the conference jointly with the ACM. We thank everyone involved in Australasian Computer Science Week for making this conference and its proceedings publication possible, and we thank CORE, SERL, our hosts AUT University, New Zealand, and the Australasian Computing Education executive for the opportunity to chair the ACE2014 conference.

Jacqueline Whalley
AUT University, Auckland

Daryl D'Souza
RMIT University, Melbourne

ACE 2014 Conference Co-chairs
January 2014
Programme Committee and Additional Referees

Chairs

Jacqueline Whalley, AUT University, New Zealand
Daryl DSouza, RMIT University, Australia

Members

David J. Barnes, University of Kent, UK
Tim Bell, University of Canterbury, New Zealand
Angela Carbone, Monash University, Australia
Michael Caspersen, Aarhus University, Denmark
Jason Ceddia, Monash University, Australia
Alison Clear, Christchurch Polytechnic Institute of Technology, New Zealand
Tony Clear, AUT University, New Zealand
Mats Daniels, Uppsala University, Sweden
Michael De Raadt, Moodle, Australia
Julian Dermody, 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, New Zealand
Raina Mason, Southern Cross University, Australia
Chris McDonald, University of Western Australia, Australia
Dale Parsons, Otago Polytechnic, New Zealand
Arnold Pears, Uppsala University, Sweden
Anne Philpott, AUT University, New Zealand
Helen Purchase, The University of Glasgow, UK
Anthony Robins, University of Otago, New Zealand
Judy Sheard, Monash University, Australia
Simon, University of Newcastle, Australia
Josh Tenenberg, University of Washington, USA
Errol Thompson, Aston University, United Kingdom

Conference Webmaster

Jacqueline Whalley, AUT University, New Zealand
Organising Committee

Chairs
Tony Clear and Russel Pears

Venue
Tony Clear

Communications
Russel Pears, Hui Ling Tan, Melanie Curry-Irons and Ryan Butler

Finance
Alison Clear and Eva Ihaia

Sponsorship
Stephen Thorpe

Operations
Adam Winship and Eva Ihaia

Programme, proceedings and booklet
Alison Clear

Catering and registration
AUT Hospitality Services
Welcome from the Organising Committee

On behalf of the Organising Committee, it is our pleasure to welcome you to Auckland and to the 2014 Australasian Computer Science Week (ACSW 2014). Auckland is New Zealand’s largest urban area with a population of nearly one and a half million people. As the centre of commerce and industry, Auckland is the most vibrant, bustling and multicultural city in New Zealand. With the largest Polynesian population in the world, this cultural influence is reflected in many different aspects of city life. ACSW 2014 will be hosted at the City Campus of Auckland University of Technology (AUT), which is situated just up from the Town Hall and the Auckland central business district. ACSW is the premier event for Computer Science researchers in Australasia. ACSW2014 consists of conferences covering a wide range of topics in Computer Science and related areas, including:

- Australasian Computer Science Conference (ACSC) (Chaired by Bruce Thomas and Dave Parry)
- Australasian Computing Education Conference (ACE) (Chaired by Jacqueline Whalley and Daryl D’Souza)
- Australasian Information Security Conference (AISC) (Chaired by Udaya Parampalli and Ian Welch)
- Australasian User Interface Conference (AUIC) (Chaired by Burkhard C. Wünsche and Stefan Marks)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Bahman Javadi and Saurabh Kumar Garg)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by James Warren)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Georg Grossmann and Motoshi Saeki)
- Australasian Web Conference (AWC) (Chaired by Andrew Trotman)

This year reflects an increased emphasis for ACSW on community building. Complementing these published technical volumes therefore, ACSW also hosts two doctoral consortia and a number of associated workshops, including those for the Heads and Professors of Computer Science, plus for the first time the ‘Australasian Women in Computing Celebration’. Naturally in addition to the technical program, there are a range of events, which aim to provide the opportunity for interactions among our participants. A welcome reception will be held in the atrium of the award winning newly built Sir Paul Reeves Building, which has integrated the city campus as a hub for student activity and provides a wonderful showcase for this year’s ACSW. The conference banquet will be held on campus in one of the reception rooms in this impressive complex.

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 ACSW2014. 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 symposia, workshops, poster sessions and accompanying conferences. Special thanks to Alex Potanin, as the steering committee chair who shared valuable experiences in organising ACSW and to John Grundy as chair of CoRE for his support for the innovations we have introduced this year. We’d also like to thank Hospitality Services from AUT, for their dedication and their efforts in conference registration, venue, catering and event organisation. 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.

We hope your stay here will be both rewarding and memorable, and encourage you to take the time while in New Zealand to see some more of our beautiful country.

Tony Clear
Russel Pears
School of Computer & Mathematical Sciences
ACSW2014 General Co-Chairs
January, 2014
CORE welcomes all delegates to ACSW2014 in Auckland. 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 have now been joined by the Australasian women in computing celebration (AWIC), two doctoral consortia (ACDC and ACE-DC) and an Australasian Early Career Researchers Workshop (AECRW) which reflect the evolving dimensions of ACSW and build on the diversity of the Australasian computing community.

In 2014, we have again chosen to feature a small number of keynote speakers from across the discipline: Anthony Robins (ACE), John Mylopoulos (APCCM), and Peter Gutmann (AISC). I thank them for their contributions to ACSW2014. 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 Tony Clear, Russel Pears and their colleagues for organising what promises to be a vibrant event. Below I outline some of CORE’s activities in 2012/13.

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 2013 this includes a new portal with a range of holistic venue information and a community update of the CORE 2009 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 Mathematics decadal plan

Coordination with other sector groups:

– Work with the ACS on curriculum and accreditation
– Work with groups such as ACDICT 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

A new Executive Committee from 2013 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, 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 2013, and look forward to the continuing shaping and development of CORE in 2014.

John Grundy
President, CORE
January, 2014
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.

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


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.
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.
1998. Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.
1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.
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.
1987. Volume 9. Host and Venue - Deakin University, VIC.
1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.
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

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDC</td>
<td>Australasian Computing Doctoral Consortium</td>
</tr>
<tr>
<td>ACE</td>
<td>Australasian Computing Education Conference</td>
</tr>
<tr>
<td>ACSC</td>
<td>Australasian Computer Science Conference</td>
</tr>
<tr>
<td>ACSW</td>
<td>Australasian Computer Science Week</td>
</tr>
<tr>
<td>ADC</td>
<td>Australasian Database Conference</td>
</tr>
<tr>
<td>AISC</td>
<td>Australasian Information Security Conference</td>
</tr>
<tr>
<td>APCCM</td>
<td>Asia-Pacific Conference on Conceptual Modelling</td>
</tr>
<tr>
<td>AUIC</td>
<td>Australasian User Interface Conference</td>
</tr>
<tr>
<td>AusPDC</td>
<td>Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)</td>
</tr>
<tr>
<td>AWC</td>
<td>Australasian Web Conference</td>
</tr>
<tr>
<td>CATS</td>
<td>Computing: Australasian Theory Symposium</td>
</tr>
<tr>
<td>HIKM</td>
<td>Australasian Workshop on Health Informatics and Knowledge Management</td>
</tr>
</tbody>
</table>

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.
ACSW and ACE 2014 Sponsors

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

**Host Sponsor**

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

**Platinum Sponsor**

DATACOM,
www.datacom.com.au

**Gold Sponsor**

The University of Auckland,
www.auckland.ac.nz

**Silver Sponsors**

COLAB - AUT Design+Creative Technologies,
colab.aut.ac.nz

Institute of IT Professionals, New Zealand,
www.iitp.org.nz

**Bronze Sponsor**

SERL - AUT Software Engineering Research
Laboratory,
www.serl.aut.ac.nz

**Australian Computer Society**, www.acs.org.au

**Publication Sponsor**

University of Western Sydney,
www.uws.edu.au

**Sponsors of ACE**

RMIT University,
www.rmit.edu.au

Association for Computing Machinery,
www.acm.org

ACM Special Interest Group on
Computer Science Education,
www.sigcse.org
CONTRIBUTED PAPERS
The Australian Digital Technologies Curriculum: Challenge and Opportunity

Katrina Falkner  Rebecca Vivian  Nickolas Falkner
School of Computer Science  The University of Adelaide  Adelaide, South Australia
Firstname.lastname@adelaide.edu.au

Abstract
There is a call for change in the treatment of ICT curriculum in our schools driven by the relatively recent acknowledgement of the growing importance of ICT in industry and society, and the need to empower youth as producers, as well as consumers, of technology. ICT curriculum in previous incarnations tended to focus on ICT as a tool, with the development of digital literacy as the key requirement. Areas such as computer science (CS) or computational thinking 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 under debate within Australia, 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, Foundation (F) to year 12.

In this paper, we discuss the main trends and learning objectives of these new curricula, identifying key areas requiring further research and development by the CS Education community. We undertake a review of current research in CS Education within the F-12 context, to identify research that can guide effective implementation and provide opportunities for further research.

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

1 Introduction
Over the last decade, the need to rethink our education systems in terms of the treatment of computer science (CS) and information technology has gained global attention (Gander et al., 2013; Seehorn et al., 2011; The Royal Society, 2012). We struggle to attract potential students and to promote CS as a creative, engaging career, despite the growing need for CS professionals. Recent US statistics indicate that only 2% of SAT takers intending to pursue college degrees intend to major in CS (College Board, 2012). The “Shut down or restart?” report by The Royal Society (2012) states: “despite the near-ubiquity of computer technology, there is now a dwindling interest in studying Computing at school”.

Considerable research has explored the reasons behind this disparity, focussing on negative career perceptions, the identity issues caused by the confusion of CS with the simplistic application of ICT tools (Schulte et al., 2012), gender differences (Henwood, 2000) and other stereotypes (Jepson & Perl, 2002).

Over the past decade we have witnessed a transition in ICT education from ICT as a tool - with the development of digital literacy as the key requirement - moving toward understanding the underpinning concepts and workings of ICT. Areas such as CS or computational thinking were typically isolated into senior secondary programs, with a focus on programming and algorithm development, when they were considered at all. Despite the recognised need for CS education, 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 and Stephenson, 2009). 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). This is a driver for CS to be taught in school, as early as the first year. Encouraging students to engage in current technologies and participate as creators of future technologies requires more than teaching the fundamentals of digital literacy – familiarity with the tools and approaches to interact with technology. We must also teach computational thinking, the problem solving processes and intellectual practices needed to understand the scientific practices that underpin technology. Without this, 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).

However, these reports stress 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, 2012). It is essential that our education systems evolve, requiring the clear articulation of CS as a distinct discipline, including integrating CS as a fundamental learning area across curricula, and exploring the societal and cultural impacts of technology.

New curriculum 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 youth in participating in an increasingly digital society. While this movement has many positive aspects, the introduction of such curricula poses many challenges for those involved: appropriate and inclusive development for teachers, research into pedagogy and approaches, and integration with current efforts in CS education that span primary-secondary education, and integration into further study.

In this paper, we provide an overview and discuss the core learning objectives of two new curriculum documents that introduce CS as a learning area: Australia’s proposed Digital Technologies curriculum and England’s computing curriculum. Additionally, we undertake a review of current research in CS Education within the primary and secondary context. Our goal is both to identify key sources of information that may be used to guide effective implementation, as well as identifying areas of research that have been insufficiently researched to date.

2 Next Generation ICT Curricula

Different terminology has been applied to identify the study of this discipline. For example computer science is used in the US (Seehorn et al., 2011), informatics in Europe (Gander et al., 2013), computing in England (Department for Education, 2013) and computational thinking or even ICT have been used in curriculum discussions. Australia introduces this learning area as the digital technologies. To demonstrate the variety of terminology, we draw on 71 articles later analysed in this paper, presenting the most frequent words (frequency increased by text size) used by authors to describe the discipline. For consistency, we have chosen to use the term computer science (CS), unless referring to particular curricula.

![Figure 1: 25 most frequent words used to describe the discipline across 71 papers](image)

2.1 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) followed by year 1, and so on, until year 6 or 7, (depending on the state) and secondary school (also known as high school) includes years 7 or 8 to year 12. In 2013, the Australian Curriculum Assessment and Reporting Authority (ACARA) released a series of draft curriculum standards for the national curriculum that is 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 F-12, while others, including ICT, have achievement standards defined from F-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 additionally introduce a new learning area, Technologies, combining the “distinct but related” areas of Design and Technologies and Digital Technologies (DT) (ACARA, 2013). DT explicitly addresses computational thinking and the use of digital systems and data, spanning representation, abstraction, algorithmic design, fundamental programming, requirements analysis and cultural impacts of technology.

An information report released by ACARA states that 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 (ACARA, 2013). 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 these year-groupings. The development of both digital literacy and computational thinking commences in the F-2 band. In F-2, learning is based around directed play, facilitating students in developing an understanding of the relationship between the real and virtual worlds, the use of technology in communication, and the importance of precise instructions and simple problem solving in the digital world. In 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. Across 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
proceeds using technology, and understand complex and abstract processes. This development from F-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:

- **F-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:

- **F-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 computational thinking and presents challenges to us as a community in how we develop relevant skills within the younger age-groups.

### 2.2 The National Curriculum in England

England’s new National Curriculum, to be introduced in 2014, places the education of computing across two main learning areas: “computing”, and the study of “design and technology”. Computing as a discipline is a required study element across the curriculum, while the study of design and technology is a required component across Stages 1-3, addressing primary and junior secondary education. At Stage 4 (years 10-12) students may elect to study an information technology topic in-depth.

**Computing:** The Computing curriculum explicitly targets the development of CS skills, including the understanding of fundamental CS concepts, the ability to analyse problems and develop computer programs to solve those problems and the evaluation of information technology solutions. At Stage 1 (years 1-2), students will have direct exposure to programming languages, including skills in creating and debugging simple programs, as well as cyber-security and digital literacy.

At Stage 2 (years 3-6), students develop more complex programming skills, including decomposition, iteration and selection, logical reasoning and error detection. At Stage 3 (years 7-9) move to a more abstract level, exploring computational abstractions that model real-world problems, sorting and searching algorithms, use of two or more programming languages, modularity and decomposition, and digital representation.

**Design and Technology:** At Stage 1 (1-2), students explore designing, making and evaluating technology, with an emphasis on physical structures and, where appropriate, ICT. At Stage 2 (3-6), digital literacy and CS become more prominent, incorporating the use of modelling tools and computer aided design, and the ability to programme in order to monitor and control products as a key technical knowledge component. At Stage 3 (7-9), this development is elaborated through elements of digital literacy (computer-based tool usage, digital presentations and modelling) and CS (applying their knowledge of computing to embed intelligence in products, with reasoning about explicit inputs and control outputs), along with a deeper understanding of the social and ethical impacts of technology, and consideration of culture and user needs within design.

### 2.3 Discussion

Both the Australian and English curricula integrate digital literacy and computational thinking from the Foundation year level. While the English curriculum focuses explicitly on programming and programming languages, the Australian curriculum introduces programming through a focus on the problem solving abilities required. In addition, the Australian curriculum introduces digital representation at an early point, with a stronger focus on understanding data. The English curriculum focuses on a stronger understanding of abstraction, and more advanced software decomposition and design methodology.

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 F-7), appropriate pedagogy, and skills needed for integration of
DT learning objectives with the teaching of other learning areas. 55% of respondents indicated concern with the manageability of the implementation of the DT curriculum, while 45% of respondents did not think that the learning objectives were realistic.

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, 2013b; Gander et al, 2012). Bell, Newton, Andreae, and 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. 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, in addressing the learning of CS or computational thinking from the Foundation year onwards, do we as a community fully understand the pedagogy that is needed? As a community, there have been many efforts over recent decades devoted to exposing pre-tertiary students to CS and programming, via initiatives such as CS4HS (Google, 2013), CS4FN (CS4FN, 2013), Georgia Computes! (Georgia Tech, 2012), or with resources like CS Unplugged (Computer Science Unplugged, 2013). These efforts are often implemented with the aim of changing stereotypes and encouraging participation in non-traditional student groups. CS is a young field, and there is much to learn about how to integrate computational thinking principles and digital literacy concepts with traditional early education pedagogy. This presents a considerable challenge to the CS Education community, but also an opportunity for us to reassess the direction of our research and explore the open research questions ahead of us.

3 Computer Science Education Research

How can we use existing findings to inform the implementation of the DT learning objectives? Our approach is to review the existing literature within CS education in the context of F-12, exploring the following questions:

- What research exists to guide teaching CS to students aging from 5 years to 18 years?
- Which methodologies have researchers used?
- Which DT concepts do the studies investigate?

4 Methodology

There have been a number of surveys examining the literature in CS education. Fincher and Petre (2004) and Pears et al. (2007) explore the different subfields within CS education research. More recently, Malmi et al. (2010) have undertaken a review characterising CS education research according to the type of research undertaken, specifically exploring associated theories and frameworks, research purpose and data collection. Sheard, Simon, Hamilton, and Lönnberg (2009) report on a survey of CS education within introductory programming, identifying common trends and limitations of the current research. They identify that investigating student learning in terms of established theories of learning are rare, and deserving of more research attention. Most relevant to this work is the methodological review of Randolph (2008) of program evaluations in F-12, published prior to 2005, which resulted in the identification of 29 reports. The majority of the evaluation reports related to US studies, and only 3 of the reports were set within the F-6 context.

We adopted Simon’s classification system as it was suitable for our purposes, has been applied to a number of computing education conferences (Simon, 2007, 2008; Simon, Carbone, et al., 2008; Simon, Sheard, et al., 2008). The approach has been validated previously with fairly consistent results, with the exception toward difficulty in identifying ‘topic’ (also referred to as ‘theme’ in Simon, Carbone, et al., 2008). In the following section we describe the instruments used and elaborate on the classification processes along with our search process.

4.1 Analysis Procedure

We have reviewed existing research papers about CS Education implemented for children between the ages of 5 and 18. We undertook a semi-systematic literature approach to review each paper 1) by classification, using Simon’s system (Simon, 2007) to determine context, topic, scope and nature; 2) identify the subject matter taught that aligns with the Australian key concepts for the Digital Technologies curriculum; 3) to identify the age group studied; and 4) to identify data collection methods reported. We used software tools EndNoteX5 and NVivo 10 to organise our classifications and to “code” papers. While Simon’s system has been broadly applied across CS-related conference proceedings, we have a particular focus on research that appear in journals and conference proceedings about CS Education for 5-18 year olds. We explain how our specifications relate to Simon’s process below and identify those that emerged in our analysis of the field in the Results section. We briefly describe each dimension in the system, however, for a thorough description of Simon’s classification see Simon (2007).

Simon’s scheme classifies papers across four dimensions, which include: topic, context, scope and nature. The topic dimension describes what the paper is about, for example ‘ability/aptitude’, ‘curriculum’ or a ‘teaching/learning tool’. The context dimension includes the subject area in which the paper is situated, such as the area of programming or group work. Where topic and context differ is that a paper may be in the area of ‘programming’, but the topic of focus is specifically student ‘aptitude/ability’. Although the previous studies have identified a number of topics and contexts covered, we intend to see those relating to CS education at the schooling level, so do not expect to see work on ‘capstone projects’ or ‘work experience’ (contexts) or ‘tutors and demonstrators’ (topics). Instead we expect the emergence of topics and contexts particular to this review. Scope describes the breadth of the paper, such as within a subject, an institution, a department/program or across multiple institutions. Many efforts to teach CS in primary and high school contexts are currently situated within initiatives, camps, or programmes inside or outside
of the classroom and so we have included another scope called ‘intensive program/initiative’. The nature dimension describes the type of paper. Simon’s classification includes four: ‘experiment’ and ‘analysis’ (which, combined constitute ‘research’ papers), ‘reports’ and ‘position’ papers. An ‘experiment’ examines a specific research question or hypothesis and collects data to test or answer the research question. An ‘analysis’ is a paper that analyses existing data and a ‘report’ is a report on something that has been done, possibly in conjunction with a basic survey. Our analysis excludes position papers as these are not fully implemented or evaluated.

We included a further classification named age band. The possible bands within this classification align with the Australian curriculum (ACARA, 2011) and include year levels grouped as: Lower primary: F-2 (ages 5-7) and 3-4 (ages 8-9), Middle: Year 5-6 (ages 10-11) and 7-8 (ages, 12-13) and Upper/HS: 9-10 (ages 14-15) and Year 11+ (16+).

Additionally, we created a broad-level classification for studies conducted across multiple year levels. Where articles targeted a specific age range or a number of age ranges, we classified according to the ‘best fit’ (e.g. for an article about ages 13-15, band 9-10 was selected).

To determine the variety of CS concepts found in the papers we used the ACARA document (ACARA, 2012, pp. 63-64) as a guide to code content identified in the papers as being the object of study in the activities being researched or reported. We created a document based on the desired key concepts on page 63-64, including a description and “content terms to look out for”. If the subject content were mentioned in the paper it was coded to the relevant ‘key concepts’ nodes in NVivo.

Methodology was another aspect of interest in our review. We coded any mention of data collection techniques to particular nodes we created in NVivo (e.g. interviews, focus groups) and classified each article as being ‘mixed’ methods or ‘qualitative’ or ‘quantitative’.

4.2 Process and procedure

We searched Google Scholar and the ACM Digital Library database for articles about the F-12 CS Education, limited to 2003-2013. Google search terms included those associated with ‘computer science’ (‘informatics’, ‘programming’, ‘computing’) and words such as ‘education’, ‘activities’, ‘learning’, and ‘lesson’. Year-level search terms used included ‘schooling’, ‘high school’, ‘primary’, ‘elementary’, ‘F-10’, ‘F-12’ and their derivatives. As the ACM Digital Library has a CS focus, we wanted to source articles with a F-12 and lesson focus and searched the database using the terms ‘school’, ‘activities’, ‘lessons’, ‘students’ and their derivatives.

Table 1: Inclusion/exclusion criteria

<table>
<thead>
<tr>
<th>Inclusion</th>
<th>Exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003-2013</td>
<td>Before 2003</td>
</tr>
<tr>
<td>F-12 (ages 5-18)</td>
<td>University/college</td>
</tr>
<tr>
<td>Research papers and reports</td>
<td>Position papers</td>
</tr>
<tr>
<td>About the implementation</td>
<td>Theoretical papers</td>
</tr>
<tr>
<td>of activities for teaching CS-</td>
<td>Teachers and PD programs (other</td>
</tr>
<tr>
<td>related concepts</td>
<td>than design and implementation</td>
</tr>
<tr>
<td>Situated within any context</td>
<td>of lessons/initiatives)</td>
</tr>
<tr>
<td>Student-focused</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Inclusion/exclusion criteria

4.2 Process and procedure

We searched Google Scholar and the ACM Digital Library database for articles about the F-12 CS Education, limited to 2003-2013. Google search terms included those associated with ‘computer science’ (‘informatics’, ‘programming’, ‘computing’) and words such as ‘education’, ‘activities’, ‘learning’, and ‘lesson’. Year-level search terms used included ‘schooling’, ‘high school’, ‘primary’, ‘elementary’, ‘F-10’, ‘F-12’ and their derivatives. As the ACM Digital Library has a CS focus, we wanted to source articles with a F-12 and lesson focus and searched the database using the terms ‘school’, ‘activities’, ‘lessons’, ‘students’ and their derivatives.

Table 1: Inclusion/exclusion criteria

Relevant papers matching our inclusion/exclusion criteria (Table 1) were entered into EndNote X5 with the PDF as an attachment. The Endnote file was exported to NVivo version 10 for classification and coding.

5 Results

The search for articles returned 71 results that matched our inclusion criteria. Table 2 describes the descriptives of the articles sourced using Simon’s classification.

5.1 Summary of research articles

Some 40 papers were reports: discussing the outcomes of a particular activity or outreach program, using researcher experiences, observation or a basic end-of-course questionnaire. 30 papers were based on experiments (or a study) where researchers used research methods to gather data to answer a particular research question. Although these were also about outreach programs or activity outcomes, the researchers used a combination or more rigorous use of methods. However, many measured student engagement or interest, rather than pedagogical effectiveness or students’ achievement. Use of existing data of students’ work was classified as ‘analysis’.

Table 3 demonstrates that the majority of studies were conducted in the United States (US; 39), followed by European regions and Asia.

Table 3: Number of articles by origin

<table>
<thead>
<tr>
<th>US</th>
<th>EU</th>
<th>Asia</th>
<th>UK</th>
<th>AU</th>
<th>NZ</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>15</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 4: Number of articles by published year

<table>
<thead>
<tr>
<th>'04</th>
<th>'05</th>
<th>'06</th>
<th>'07</th>
<th>'08</th>
<th>'09</th>
<th>'10</th>
<th>'11</th>
<th>'12</th>
<th>'13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>10</td>
<td>16</td>
<td>17</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Number of articles by published year

The scope of the paper identifies the range of the sample and context in which the paper describes. We present a cross-tabulation of context and scope in Table 6. Although university institutions run many of the initiatives and research, we can see that most of the articles were about intensive programs and initiatives, so we created a category to recognise this. A number also specifically referred to research that was conducted within a single case study so we classified these as ‘single cases’.
Other common methods involved the collection of engagement and interest after the activity or intervention, such as questionnaires and interviews, measuring student work that was examined or analysed, usually in the form of student games that they had programmed.

The most commonly used methods were qualitative methods only (32) and quantitative methods (4). Some papers used more than one method. ‘Other methods’ included collecting data by involving the students as researchers, for example, by producing journals about their processes. The most commonly used methods were questionnaires and interviews, measuring student engagement and interest after the activity or intervention. Other common methods involved the collection of student work that was examined or analysed, usually in the form of student games that they had programmed.

Table 5 presents a cross-tabulation of topics and contexts for the papers analysed. The table indicates that, similar to previous analysis using Simon’s classification with CS education research, these research papers were also most commonly situated within a ‘programming’ context. Within this context, the papers explored topics such as students’ ability or aptitude to do programming activities or the extent they applied CS concepts and knowledge to their programming. Other topics included exploring teaching and learning techniques for CS concepts or delivery of activities, trialling new teaching and learning tools and student perception and interest in programming. Other popular contexts included integrating CS within other learning areas, such as the Humanities.

In Table 8 we grouped articles by year level bands to allow the examination of types of paper topics explored within each band. From Most articles addressed children in the middle school or high school. In these year levels, the articles focused on student perceptions about doing CS activities, their ability to undertake CS tasks and teaching and learning techniques used within these age groups. Minimal research exists about students in the lower primary levels but for those articles we did source, investigated whether young children could engage in programming or computational thinking and also explored new tools that could be used to teach CS activities for children.

Table 8: Topic compared to year level band

5.2 Research Topics

Table 5 presents a cross-tabulation of topics and contexts for the papers analysed. The table indicates that, similar to previous analysis using Simon’s classification with CS education research, these research papers were also most commonly situated within a ‘programming’ context. Within this context, the papers explored topics such as students’ ability or aptitude to do programming activities or the extent they applied CS concepts and knowledge to their programming. Other topics included exploring teaching and learning techniques for CS concepts or delivery of activities, trialling new teaching and learning tools and student perception and interest in programming. Other popular contexts included integrating CS within other learning areas, such as the Humanities.

In Table 8 we grouped articles by year level bands to allow the examination of types of paper topics explored within each band. From Most articles addressed children in the middle school or high school. In these year levels, the articles focused on student perceptions about doing CS activities, their ability to undertake CS tasks and teaching and learning techniques used within these age groups. Minimal research exists about students in the lower primary levels but for those articles we did source, investigated whether young children could engage in programming or computational thinking and also explored new tools that could be used to teach CS activities for children.
Table 9 demonstrates that the majority of F-7 research within CS addresses the use of tangible programming tools (21), followed by the use of existing CS activities (with all but one involving the use of CS Unplugged; the other a German version Informatik erLeben). Scratch makes up the majority, with 10 cases, followed by three research papers examining the use of Alice.

An examination of the articles according to the DT key concepts (ACARA, 2012) in Table 10, reveals that most of the articles implemented activities that involved algorithms, implementation and specification: essentially those involved through teaching programming activities. Another commonly taught topic was data representation and interpretation. In the younger years, this involved activities such as understanding binary through tactile games or in the upper years it extended to more complex activities such as manipulating digital images. Some of the papers also discussed multiple topics within one article and we coded these as broad-based. These typically involved reporting on the success of a set of activities that covered many of the DT concepts.

| Communication of Problems and Solutions | 29 |
| Algorithms (following and describing) | 34 |
| Specification (descriptions and techniques) | 18 |
| Data | |
| Data collection (properties, sources and data collection) | 3 |
| Data interpretation (patterns and context) | 8 |
| Data representation (symbolism and separation) | 12 |
| Digital systems (hardware, software, and networks on the Internet) | |
| Hardware and software | 7 |
| Networks and the Internet | 9 |
| Interactions (people and digital systems, data and processes) | 8 |
| Broad-based concepts | 8 |
| Abstraction (hiding irrelevant details) | 7 |
| Impact (impacts and empowerment) | 1 |

Table 10: Articles according to DT key concepts

There was only one article that explored CS careers. With the Australian and English curricula addressing social and ethical impact, research is required that investigates the teaching of such content, in addition to programming skills and computational thinking. This will also be an important area for consideration if we are to make computational thinking and programming activities relevant to the lives and future careers of students.

6 Limitations

We acknowledge that we have not possibly captured all existing literature about CS education in years F-12. In our initial study, we provide a preliminary guide to current existing research and trial our analysis approach so that we can review and implement our approach on a larger scale. Our future work will continue to refine and expand databases and terms.

We also realise that CS Education research may exist within other discipline areas that were not discovered in searches, such as society and environment, design and technology, mathematics, or science because of its versatile nature and the ability for CS concepts and approaches to be applicable in other fields, as we saw with the use of programming as a tool for story telling and learning about storytelling (Burke & Kafai, 2010).

This offers opportunities for future research to identify cross-curricula use of CS within other learning areas.

Similar to Simon, Carbone, et al. (2008), we also encountered difficulties in deciding the context and topic of papers, however, using Simon’s suggestion, we made our decision on what topic or context was the ‘best fit’ when more than one possible topic existed. We acknowledge that others may classify some papers within different areas, but nonetheless the classification still provide sound guidance for what research currently exists in F-12 CS Education and what research is required.

7 Discussion

After review, three significant areas emerged that provide guidance for future research. We will discuss how these guide approaches to future research and lead into the conclusion.

7.1 CS F-10 Pedagogy

While there has been considerable research into CS within the F-12 context, it is typically focussed on years 5-12 with much less research at the F-4 level. 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. Computer games and the creation of games through tangible programming tools also play a significant role in current approaches to engaging younger students in CS, however as highlighted by Denner (2011), the majority of studies in this domain explore the potential for computer games to motivate students to study CS, rather than exploring what they are able to learn. This is of increasing importance with the emerging focus on computational thinking and the development of computational problem solving skills.

There is a whole field of possibilities for pedagogical exploration in F-10 CS education and to investigate specific techniques for early education within CS, including small-group ability levels, inquiry-based learning, and play-based learning.

Compare this with the field of Mathematics education, with its rich history of deep exploration of Mathematics pedagogy. Some interesting recent examples that highlight potential areas for related CS research include: analysis of symbolic number sense and impact upon mathematics achievement (Jordan et al, 2009); analysis of core concepts and student understanding (Knuth et al, 2011); gender-based stereotypes and achievement (Beilock et al, 2010); and emergent mathematical thinking in play environments (van Oers, 2010).

Similarly, there are opportunities for exploring how the use of CS tools influences learning processes. Papert’s (1980) work in programming environments for children introduces the idea of constructionist programming environments: places where children can create concrete digital constructs from abstract ideas, and then reflect over those to develop understanding. Many of the constructionist programming environments are focused on years 3-7, including Scratch, Alice and Kodu. In this emerging field, there is early work that demonstrates that children who are exposed to constructionist environments are able to learn
computational thinking concepts (Bers & Horn, 2010; Kazakoff & Bers, 2012; Kazakoff, Sullivan, & Bers, 2013; Lai & Yang, 2011). In contrast, a study by Meerbaum-Salant, Armoni, and Ben-Ari (2011) identifies that use of Scratch engenders specific poor programming habits, at odds with both accepted practice and the learning objectives of the proposed curricula.

This also applied to the lesson resources that currently exist, such as CS Unplugged. These resources are helpful, especially for teachers who have limited or no experience in CS and are able to be implemented in classrooms with no technology. However, we must be clear on the goals of a program such as CS Unplugged. Taub, Ben-Ari, and Armoni (2009) state the three main aims of CS Unplugged as changing students’ views on the nature of CS, promoting views that CS is a career for women and changing views about CS as a profession. An analysis of the CS unplugged resources to determine approach, coverage of explicitly addressed CS concepts and whether the aims were addressed identified that only some of the objectives were addressed in the activities. After trialing activities, year 7 students did change their understanding of the nature of CS, but held less attractive perceptions of CS as a career. Similarly, Feaster, Segars, Wahba, and Hallstrom (2011) found implementation of a semester long outreach program using the resources had no significant impact on attitudes toward CS or content understanding. Once again, activities like CS Unplugged have typically been assessed in terms of their effectiveness to change attitudes and perceptions, rather than learning progress. There are new opportunities for evaluating existing CS activities in terms of student achievement, learning objectives and improved computational thinking processes.

7.2 Methodology, Sample and Scope

Many studies were conducted with small sample sizes or were pilot studies due to being situated within the work of intensive programs or initiatives and because many were about show-casing and sharing teaching and learning techniques or tools (Kordaki, 2011; Lewis, 2011). Furthermore, the studies are usually conducted outside of conventional classroom settings and authors identify that it is difficult to make a comparison to classroom environments (Lode, Franchi, & Frederiksen, 2013). If studies were conducted in-class they were typically one-off sessions, out of the context of the regular curriculum, which authors cautioned may have result in students and teachers being ‘less committed’ (Meerbaum-Salant, Armoni, & Ben-Ari, 2010).

Another limitation was that students who were the subject of study were usually involved because they volunteered to participate in after school or holiday programs (Denner, Werner, & Ortiz, 2012; Lau, Ngai, Chan, & Cheung, 2009; Magenat, Riedo, Bonani, & Mondada, 2012). As volunteers, the participants may come to the classes out of interest: a different frame of mind to students who are in classrooms out of duty. Other studies selected students based on their achievement, for example in a study by Curzon, McOwan, Cutts, and Bell (2009) participants were identified as being in the top 5% of the school and participants in research by Feaster, Ali, and Hallstrom (2012) involved high achievers. In classroom environments, teachers typically have to cater to students with a whole range of capabilities, interests and achievement levels making this a challenge for teachers to overcome.

The actual effectiveness of teaching techniques are often not known because researchers have not measured before and after (Meyers, Cole, Korth, & Pluta, 2009) and because researchers experienced difficulty in identifying ways to formally assess goals and outcomes of projects (Settle et al., 2012). Ultimately, research in this area will need to be rigorous, replicable and explicitly defined.

7.3 Teacher Experiences and Development

Our review of the literature was focused on students and the implementation of the lessons, rather than teacher ability and training, but one important aspect that arose was in regard to who was implementing the activities that were the object of study. In many cases, activities were conducted by researchers from CS institutions or by those with significant experience in teaching CS. For example, in Meerbaum-Salant et al. (2011) the teacher had 15 years experience with teaching CS and in Taub et al. (2009) one teacher taught mathematics and programming and the other teacher had one year’s experience teaching CS Unplugged. Robertson & Nicholson 2007 involved a specialist IT teacher and three researchers; and in a study by Stoeckelmayr, Tesar, and Hofmann (2011) the activity was conducted by a CS academic from a university with the support of undergraduate students. These situations are vastly different to a single generalist teacher implementing classroom activities without support.

Authors, Settle et al. (2012), recognise the difficulty in translating materials into existing curriculum, when unfamiliar with the tools. 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). The teachers also felt that because they were unfamiliar with the topic, considerable preparation would be required. Meerbaum-Salant et al (2011) identified that although the teacher was experienced in CS, adding new tools created anxiety, causing deviation from lesson plans. Tinapple, Sadauskas, and Olson (2013) further comment on the challenge for teachers, where expected software and/or hardware are not easily available.

Black et al. (2013) describe a survey of UK computing teachers in relation to their suggestions on improving CS education, and teacher development needs. Their results highlighted teacher training, and the need for a network and community to support resource development. Black et al’s survey identifies that teachers focus more on fun activities rather than providing opportunities for deep learning of computational thinking, 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.

8 Conclusions

The expected changes in the teaching of Computer Science represent a significant challenge for our schooling systems. Computational Thinking and
Computer Science will form part of the Australian standard curriculum from F-12 from 2014. In this paper, we have presented the key learning objectives of both curricula, and have identified the key challenges that arise from these changes, specifically, the need to teach computational thinking as a standalone concept; the introduction of computational thinking and computer science from Foundation onwards, and the need to develop and understand appropriate pedagogy that integrates with existing early childhood approaches.

We have undertaken a preliminary review of existing CS education research within the F-12 context, identifying key themes (outreach, programming, tangible programming tools, CS activities, senior secondary) and also gaps (F-7, computational thinking, CS concepts). We have identified a distinct lack of rigorous research within the F-7 context, including relevant pedagogy and assessment practices within conventional classroom settings. This represents an outline of needed research requiring greater collaboration between representatives in primary and secondary school education, education researchers, and higher education CS departments. With greater collaboration between each group it may better ensure the development of a research agenda that encompasses the expertise and needs of both groups.

9 References


Visualising Career Progression for ICT Professionals and the implications for ICT Curriculum Design in Higher Education

Brian R. von Konsky  
Curtin Teaching and Learning  
Curtin University  
Perth, Western Australia  
B.vonKonsky@curtin.edu.au  

Asheley Jones  
ACS Education  
Australian Computer Society  
Melbourne, Victoria  
asheley.jones@acseducation.edu.au  

Charlynn Miller  
Science, IT & Engineering  
University of Ballarat  
Ballarat, Victoria  
c.miller@ballarat.edu.au  

Abstract
The current environment in higher education calls for more consideration of the linkages between ICT curriculum development, skills capabilities and industry, particularly in light of recent changes in quality and standards agencies. This paper evaluates ICT career progression visualisation methodology and has a threefold purpose: to contribute to a holistic approach to curriculum design and management; to add to materials that aid graduates to better prepare initial professional practice choices for employment in the ICT profession; and to facilitate further dialogue with industry representatives, higher education providers and other ICT stakeholders to ensure undergraduate curricula authentically reflects the skills required within the ICT profession. This paper evaluates SFIA-based tools intended to enable educational designers to visualise ICT career progression pathways and thus inform curriculum design in higher education. Several visualisation techniques are compared using SFIA-based skillsets that were previously published in the literature. The evaluation demonstrates extended radar diagrams are an effective visual representation for capturing the level at which SFIA skill sets are practiced. The research indicates that such representations are well positioned to enhance dialogue amongst stakeholders and contribute to the design of ICT curriculum in a manner that better prepares students for ongoing development in the profession.

Keywords: SFIA, curriculum, ICT education, professional practice, skills, competencies

1 Introduction
ICT education worldwide is in flux, partly as a result of the imperative to better align educational curriculum with industry needs. The Skills Framework for the Information Age (SFIA) is a dynamic two-dimensional skills matrix managed by the SFIA Foundation, a consortium formed in July 2003 by the Institution of Engineering and Technology (IET), Institute for the Management of Information Systems (IMIS), e-skills UK, and the British Computer Society (BCS). This reference model, now in its fifth iteration, can be used for describing Information Communication Technology (ICT) skills and the levels of responsibility at which they are practiced (SFIA Foundation, 2011a). Similarly, the framework can be used by organisations providing ICT products and services as a standardised means by which to manage the recruitment, assessment, and development of ICT professionals (SFIA Foundation, 2011b).

The Australian Computer Society (ACS) also uses SFIA for member certification (ACS, 2012b) and the accreditation of higher education programs that prepare students for initial professional practice in the ICT industry (ACS, 2012a). ACS recommends that higher education institutions adopt a top-down approach to curriculum design that begins with SFIA to define ICT career roles for which a given program prepares graduates (ACS, 2012a, 2012c). Using such an approach, SFIA has been embedded in the ACS Computer Professional Education Program (CPeP). This is a postgraduate program that offers an articulation pathway to a number of Australian masters programs.

This paper examines techniques to visualise SFIA skillsets along a career path that includes: those skills developed in undergraduate ICT programs; and in positions held by graduates in the early stages of their career development. A goal is to take a holistic approach to curriculum design and management, such that graduates are adequately prepared for initial professional practice in the ICT industry. A further goal is to facilitate dialogue with industry representatives and other stakeholders to ensure that the undergraduate curriculum authentically reflects the skills required by industry.

2 Background
SFIA defines generic attributes that encompass business skills and the extent to which an individual demonstrates autonomy and influence. These are defined across the 7 levels of responsibility that are shown in Table 1 (SFIA Foundation, 2011a, 2011b).

<table>
<thead>
<tr>
<th>Type</th>
<th>Characterisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Follow</td>
</tr>
<tr>
<td>2</td>
<td>Assist</td>
</tr>
<tr>
<td>3</td>
<td>Apply</td>
</tr>
<tr>
<td>4</td>
<td>Enable</td>
</tr>
<tr>
<td>5</td>
<td>Ensure, Advise</td>
</tr>
<tr>
<td>6</td>
<td>Initiate, influence</td>
</tr>
<tr>
<td>7</td>
<td>Set strategy, inspire, mobilise</td>
</tr>
</tbody>
</table>

Table 1: SFIA Levels

The standard also defines 96 ICT specific skills in 6 categories and 19 sub-categories. Descriptors are provided for each skill and for each of the 7 responsibility levels at which a skill is defined.

Copyright (c) 2014, Australian Computer Society, Inc. This paper appeared at the 16th Australasian Computing Education Conference (ACE2014), Auckland, New Zealand, January 2014. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 148. J. Whalley and D. D'Souza, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.
Not all skills are defined for each of the 7 levels. For example, the Service Desk and Incident Management (USUP) skill is practiced between Levels 1 (follow) and Level 5 (ensure, advise). At the low end of this range, an individual handles customer support and associated record keeping, usually under the direct supervision of a more senior colleague. At the higher end of this range, an individual maintains policies and standards associated with the provision of client support. In contrast, the Consultancy (CNSL) skill is only defined at the higher end of the responsibility range from Levels 5 (Ensure, advise) to Level 7 (set strategy, inspire, mobilise). At Level 5, a consultant is responsible for understanding user requirements, collecting data, analysing results, and delivering solutions to clients. At Level 7, a consultant operates with significant autonomy and responsibility in the provision of a wide range of consulting services. The reader is referred to the SFIA documentation for specific descriptors associated with these roles and levels (SFIA Foundation, 2011a).

In designing ICT curricula, it is possible to demonstrate the alignment of SFIA categories to knowledge areas in the ACS Core Body of Knowledge (ACS, 2012c). Similarly, it is possible to align professional competencies and skills to institutional graduate attributes used in curriculum maps that link learning experiences and assessments to the intended learning outcomes (Oliver, Jones, Ferns, & Tucker, 2007). However, potential complications include: graduate attributes that overlap with respect to the set of professional skills and competencies that vary in number in a manner that makes tabular representations problematic (Oliver, 2013).

None-the-less, the use of tables to document course structures and their alignment with specified knowledge areas and skills are commonplace. This includes the use of tables associated with course accreditation. For example, the ACS provides standard forms with tables to be used by institutions requesting accreditation. These forms show the program structure and its alignment with knowledge areas from the ACS CBOK.

The University of Tasmania has used a tabular approach to identify SFIA skills to be embedded in new ICT curriculum that is currently being designed for implementation commencing in 2014 (Herbert, de Salas, et al., 2013). It is important to note that aligning a course with SFIA is subtly different from embedding these skills directly in the curriculum (Bailey, 2012). That is, embedding SFIA in the curriculum should not be a “tie the box” exercise to demonstrate compliance. Rather, SFIA should directly inform the design and structure of learning activities and add value to the learning experience with respect to their authenticity and relevance to industry.

Tabular curriculum maps have also been augmented with geometric symbols to convey further information about the mapping. For example, Spencer, Riddle and Knewstub (2011) added geometric symbols of varying size and colour to produce heat maps. These symbols capture the confidence and quality of evidence that a given graduate attribute has been taught, practiced and assessed. This approach provides a visual representation of current practice and indicates where curriculum redesign activities should be focused in further design interactions.

Tabular maps based on Quality Function Deployment (QFD) with House of Quality (HoQ) graphical features, adapted from use in industrial applications, have been described in the curriculum design of a hypothetical Master of Information Systems course (Denton, Virginia Franke, & Nanda, 2005). Tables using this approach contain graphical symbols in five areas that link graduate attributes to: a common body of knowledge or skillset; the expectations of prospective employers; critical prerequisite areas; feedback received from program graduates and employers; and the relative amount of time devoted to each knowledge area.

Other visualisation approaches to convey skills and competencies have also been reported in the literature. For example, Armstrong (2011) used a network diagram to visually decompose the entire SFIA framework by category and subcategory. A network diagram provides a good visual representation of the overall hierarchy of the SFIA framework, but does not convey the specific set of skills or the level at which they are practiced for a given career role. To facilitate the latter, Armstrong proposed the use of a two-dimensional grid that uses colour-coded cells to denote the level at which a given skill is practiced. He argued that this approach provides for a visual representation in which a number of applicants for an ICT position could be quickly compared against a position description that was similarly encoded.

von Konsky, Hay and Hart (2008) used SFIA-based radar diagrams to compare advertised ICT industry positions at various levels of seniority with those developed in an undergraduate software engineering program. Visualising career paths using this approach required considering multiple spider diagrams for each educational and professional development program and industry position along a given career path. Given the large number of SFIA skills defined in the framework, this adds to the cognitive load associated with such a visual analysis as the viewer switches between diagrams.

A visual approach to demonstrate the transition from formal study to junior and then significantly more senior positions in the ICT industry has been developed by the Queensland government (Queensland Government Chief Information Office, 2013a). This approach divides concentric rings into four quadrants that categorise ICT career roles according to those associated with technology/application building, technology services, enterprise implementation, and enterprise governance. The outer ring denotes the type of formal study required for entry to roles in a given quadrant (e.g. high school, TAFE or university qualification or industry experience). The next ring denotes roles associated with junior and first-line management positions. The inner ring contains very senior ICT career roles, whilst the centre of the diagram is limited to the Chief Information Officer and Chief Executive Officer roles. Each role listed in the diagram is hyperlinked to tables containing the SFIA skills and levels of responsibility associated with that role (Queensland Government Chief Information Office, 2013b). Visualising career progression from study and from role to role requires moving back and forth between
the figure and the hyperlinked tables containing the SFIA skillset associated with each role.

ICT career roles defined by the Queensland government have been used to facilitate dialogue amongst stakeholders in the design of a new undergraduate degree (Herbert, de Salas, et al., 2013). Input from industry representatives included an evaluation as to whether the Queensland government defined ICT career roles that were relevant to their organisation, and whether they had hired or would hire graduates possessing the skills associated with each role. A goal of the process was to ensure that graduates of the new program would be industry-ready for initial professional practice in the intended roles, or partially prepared to assume other roles pending additional development. Although this approach was based on an evaluation of initial skills required by graduates in industry, the process did not entail visualising career progression into the latter roles.

The principal contribution of the current paper is to evaluate approaches to visualising the career progression of ICT professionals beginning with the outcomes associated with undergraduate degree programs, and as a further aid to enhance curriculum design and inform meaningful interaction with industry stakeholders.

3 Methodology

A web application was written to enable SFIA skillsets to be combined and visualised based on published data available from multiple sources. This included intended skillsets from an undergraduate program in software engineering, ICT positions from industry at various levels of responsibility and seniority (von Konsky et al., 2008), and postgraduate subjects that develop SFIA skills in conjunction with the ACS CPeP (ACS, 2013a).

Multiple visualisation techniques were compared to evaluate their suitability for use in planning career progression, taking note of the compactness of visual representation and the range of levels defined by SFIA for each skill. This included tabular representations, radar diagrams, and extended radar diagrams.

Tabular representations were loosely based on an A3 poster published by the SFIA foundation. This poster lists all 96 skills in the framework on the vertical axis, along with a visual representation of the levels at which each skill is defined. These were colour-coded in the table by SFIA category using the same colour scheme used in the Foundation’s A3 poster. The horizontal axis contained a column for each skillset in the sample and indicated the level at which a given skill is practiced. Table cells were blank if a given skill was not included in a skillset. It was assumed that a given skill is practiced at a single level for each skillset. That is, an undergraduate program intends to develop a skill to a specified level. It is not intended to develop the same skill at multiple levels. However, a subsequent skillset for a postgraduate program might develop the skill to a higher level.

Radar diagrams were based on those used by von Konsky et al. (2008). In that approach, background cells were colour coded by SFIA category. However, the original approach was modified so that radar bars originated a fixed distance from the centre to avoid artefacts associated with small cells near the origin.

Extended radar diagrams used colour-coded cells to represent the levels for which a SFIA skill is defined in the framework. This is similar to those used in the tabular format except they are displayed radially. If a skill at a given level was included in the skillset, it was represented in the corresponding cell using a brighter shade of the colour used to represent that category. If a level is not defined in the SFIA framework for a given skill, the cell for that skill and level was not coloured.

4 Results

Figure 1 shows results in the tabular format for the following skillsets: intended skills developed in an undergraduate software engineering program (BEng (SE)); skills identified in advertised positions in industry for a Graduate Software Engineer (SE Grad) and a Software Manager (SW Mgr); and postgraduate units in the ACS CPeP including: Risk Management; Professionalism and Compliance (RMP); Business Strategy and ICT (BST); New Technology Alignment (NTA); and Business Analysis (BAS).

Intended SFIA skills developed by core subjects from an undergraduate software engineering program are shown using a radar diagram in Figure 2. It demonstrates that most of the intended skills are in the Solution Development and Implementation category (yellow). This skill category includes specific skills such as: Programming/Software Development (PROG); Testing (TEST); Safety Engineering (SFEN); Data Analysis (DTAN); System Design (DESN); Database/Repository Design (DBDS); and System Integration (SINT). Skills developed in the Strategy and Architecture category (red) include: Solution Architecture (ARCH); Methods and Tools (METL); and Software Development Process Improvement (SPIM). Additionally, the Configuration Management (CFMG) skill from the Service Management category (brown), the Quality Assurance skill from the Procurement and Management Support/Quality and Conformance category (blue) are also intended to be developed in this program. Further, the figure shows that the intended level to which these skills are developed range from Level 2 (assist) to Level 4 (enable). There are no skills in the skillset from the Business Change (purple) or Client Interface (green) categories. Complete descriptors for each skill and level can be found in documentation available from the SFIA foundation (SFIA Foundation, 2011a). It should be noted that the skillset shown in Figure 2 is for a particular software engineering program as previously reported in the literature, and is therefore not intended to characterise all undergraduate software engineering programs.

The data for an undergraduate software engineering program is also shown in Figure 3 using an extended radar diagram. Unlike Figure 2, the extended radar diagram shows the range of levels for which SFIA defines each skill in the framework.
Strategy and architecture
Business change
Solution development and implementation
Service management
Procurement and management support
Client interface
Figure 4. Skills for an advertised Graduate Software Engineer position

Figure 5. Skills for an advertised Software Manager position

Figure 6. Progression from Graduate Software Engineering to Software Manager

Figure 7. Postgraduate skills developed in ACS CPeP core subjects with a Business Analysis elective

Figure 4 through 6 are extended radar diagrams showing SFIA skills associated with advertised positions from industry. Figure 4 shows skills for a Graduate Software Engineering position. Not unexpectedly, the skills associated with this entry-level position are similar to those of the undergraduate software engineering program in that most of the skills are from the Solution Development and Implementation category. However, the Graduate Software Engineering skillset includes the Database Administration (DBAD) skill from the Services Management category. The skillset also includes Sales Support (SSUP) from the Client Interface category and the Service Desk and Service Desk and Incident Management (USUP) skill from the Services Management category.
Management categories, both at Level 1. That is, this graduate position includes low-level client facing responsibilities even though the role is largely development focussed.

As shown in Figure 5, the skillset for the Software Manager position continues to be focused strongly in the Solution Development and Implementation category. Compared to the Graduate Software Engineer, however, the Software Manager position requires a richer set of skills from this category. Moreover, they are generally performed at a higher level of responsibility. The Software Manager position also requires skills from the Business Change and Strategy and Architecture categories. Skills include: Professional Development (PDSV); Stakeholder Relationship Management (RLMT); Portfolio, Programme and Project Support (PROF); Project Management (POMG); and Business Risk Management (BRUM).

Figure 6 combines the Graduate Software Engineer and the Software Manager skillsets into a single image. Combining the skillsets in this way visually demonstrates that the Software Manager position requires growth and development with respect to the level at which some skills are practiced. For example, the visualisation shows that both the System Integration (SINT) and Systems Installation/ Decommissioning (HISIN) skills advance from Level 2 (assist) to Level 5 (ensure, advise). Similarly Testing (TEST) advances from Level 2 (assist) to Level 4 (enable).

Figure 7 shows the intended SFIA skillset associated with the ACS CPeP. The skillset for this postgraduate program includes skills developed by three core subjects and an elective subject called Business Analysis. All skills in this set are from the Business Change and Strategy and Architecture categories only. It is intended for skills to be developed at Level 5 (ensure, advise) or Level 6 (initiate, influence).

As demonstrated in the Figures 1 through 7, extended radar diagrams have a compact representation, capture the range of levels defined for each skill in the framework, and can infer skills progression at increasing levels of responsibility. In comparison, radar diagrams do not capture defined levels for each skill. Tables can capture this range, but are longer and less compact in nature.

5 Discussion
The Software Manager position requires practicing skills with additional responsibilities above and beyond those required for the graduate position. The position also includes the addition of new skills from other SFIA categories. While there are many potential pathways that a Graduate Software Engineer may take to add skills from the missing categories, one pathway includes completion of postgraduate programs such as ACS CPeP.

That is, the Graduate Software Engineer position possesses skills from the Solution Development and Implementation, Service Management, and Client Interface categories only. The position does not require any skills from the Strategy and Architecture, Business Change, or Procurement and Management Support categories. Core ACS CPeP subjects and the Business Analysis elective develop skills from the Strategy and Architecture and Business Change categories, while the Green Computing elective (not shown) adds skills from both the Strategy and Architecture and the Procurement Management Support categories. This suggests that postgraduate study is one possible path to the more senior position since it adds significant skills from the missing SFIA categories. However, this observation comes with the caveat that the specific skills developed depend on the program and choice of electives.

It is also worth noting that the undergraduate software engineering program intends to develop skills that are generally practiced at a lower level of responsibility compared to those of the postgraduate CPeP.

For example, the four-year undergraduate program develops 12 SFIA skills from 4 categories. Of these, 2 skills are at Level 2 (assist), 3 skills are at Level 3 (apply), 5 skills are at Level 4 (enable), and only 2 skills are at Level 5 (ensure, advise). The focus of the program is clearly on skills from Solution Development and Implementation category, which comprises 7 of the 12 skills in the set.

In contrast, the shorter ACS CPeP develops 9 skills from 2 categories. Of these, 8 skills are at Level 5 (ensure, advise), and 1 is at Level 6 (initiate, influence).

It is also worth comparing the skillsets for the Graduate Software Engineering position with the undergraduate software engineering program as shown in Figures 3 and 4. These skillsets are aligned in the sense that both focus on solution development and implementation. However, it is not a perfect match. For example, the undergraduate software engineering program does develop some skills from the Strategy and Architecture category that are useful for those in the Software Manager role, but not necessarily the graduate level position. Does that mean that the time spent developing Strategy and Architecture skills in an undergraduate software engineering program could be better spent developing skills required for the graduate position? This answer is “probably not”. Software architecture is a component of the Software Engineering Body of Knowledge (IEEE Computer Society, 2004). It is therefore appropriate that this be included in a course that prepares students for a role in software engineering. Even if an undergraduate program was preparing students for some other ICT role, it is reasonable to argue that exposure to the breadth of SFIA skill categories lays an appropriate foundation for future growth and development. It is further reasonable to suggest that developing depth for important skills in a specific SFIA category associated with the intended professional role is also necessary; hence the focus on solution development and implementation in this particular example.
ACS expects that academic institutions will undertake curriculum design in consultation with external stakeholders (ACS, 2012a), which often takes the form of advisory boards and focus groups that include representatives from the ICT industry (Herbert, de Salas, et al., 2013; Herbert, Dermoudy, et al., 2013; von Konsky, 2008). As these stakeholders increasingly use SFIA to identify the skills they require in the ICT professionals they employ (Banks, 2010), it makes sense that academic consultation with external stakeholders will also be based on SFIA. Moreover, SFIA-based visualisations have the potential to facilitate a shared understanding of the skills required by industry and potential career paths for early career ICT professionals as illustrated in Figure 8.

This visualisation has the potential to change processes for curriculum mapping offering a more comprehensive first phase career mapping exercise that directly links academic programs to positions in industry at increasingly higher levels of seniority.

It should not be overlooked that students and early career ICT professionals are also stakeholders in such a process. Beginning with their choice of an undergraduate program and the electives they choose, SFIA and SFIA-based visualisations have the potential to inform the decisions made by early career ICT professionals as they plan their professional development (Figure 8). Career mapping visualisation techniques such as those provided in this paper can also influence students’ choice of co-curricular and extra-curricular activities such as those sponsored by the ACS which include: monthly branch forums; the Young IT and ACS Women programs; and involvement within special interest groups (SIGs).

In the ACS CPeP program, students maintain evidence of SFIA skills in an electronic portfolio and keep an online journal in which they reflect and evidence their attainment of SFIA skills and the level of responsibility at which they practice currently (Jones & Lindley, 2010; Jones & Miller, 2012). These electronic portfolios are collated by students and assessed by a mentor. The intended learning objective from this process is to inform ongoing professional development once students successfully complete the program and become Certified Professional members of the ACS. Although CPeP is a postgraduate program, such an approach may also be applicable in undergraduate settings.

Not dissimilarly, the ACS requires that members with Certified Professional status undertake 30 hours of professional development annually. This is logged electronically on the ACS web site and can be linked to SFIA skills and levels. SFIA assessment is also available to members via the MySFIA tool, which includes a radar diagram representing SFIA skills and their level of attainment (ACS, 2013b).

Finally, it should be noted that aligning the ICT curriculum with industry-based positions using SFIA is not incompatible with the development of so-called “soft skills”. These include communication and lifelong learning. Many SFIA skills recognise that the ICT profession can change quickly. This requires that professionals be agile in their response to new technologies as they emerge, and be able to communicate the impact of emerging technology to stakeholders. For example, this is well described in the Emerging Technology monitoring (EMRG) skill.

6 Conclusions

The object of this study was threefold: to contribute to a holistic approach to curriculum design and management; to add to materials that aid graduates to better prepare initial professional practice choices for employment in the ICT profession; and to facilitate further dialogue with industry representatives, higher education providers and other ICT stakeholders to ensure undergraduate curricula authentically reflects the skills required within the ICT profession.

In order to achieve the objectives, this study has evaluated and compared various visualisation techniques for the representation of SFIA skillsets. An examination of the efficacy of these visualisation tools, specifically: tabular formats; radar diagrams; and extended radar diagrams was undertaken. The paper has demonstrated that extended radar diagrams can be seen to enable compact representation of current skill sets, with the added potential to show career progression when skillsets from different courses and positions are combined. This paper has argued that such SFIA-based tools are an important component in informing curriculum design in higher education. Visual representations such as the diagrams used to support this research have the potential to facilitate interaction and dialogue amongst stakeholders and better prepare students for ongoing development in the ICT profession. The extended radar diagrams methodology offered in this paper provides a common understanding and an opportunity for further research into holistic curriculum design. The authors of this paper expect the outcomes from the development of these models can begin to address the requirements of ICT curriculum developers in aligning learning outcomes in higher education with the needs of the ICT industry.
References


ICT Curriculum and Course Structure: the Great Balancing Act

Nicole Herbert, Kristy de Salas, Ian Lewis, Julian Dermoudy, Leonie Ellis
School of Computing and Information Systems
University of Tasmania
Private Bag 87, Hobart 7001, Tasmania
Nicole.Herbert@utas.edu.au

Abstract
This paper reports on an ICT curriculum development process that involved balancing a number of constraints that, in the words of an external academic advisory panel, resulted in a “very coherent, strong, contemporary” ICT curriculum. Instigated by an external school review that recommended the implementation of a single degree, the curriculum had to contain the knowledge requirements for students to develop the necessary skills for a set of ICT graduate level career outcomes identified by the local and national ICT industry. Due to a shrinking staff profile coupled with pressure for increased research output the School was instructed to offer only thirty undergraduate coursework units. Finally, the curriculum and course structure had to be attractive to domestic and international applicants and the curriculum also had to inspire graduate progression to a research higher degree.

Keywords: ICT career outcomes, ICT curriculum

1 Introduction
The School of Computing and Information Systems, as the only Information and Communication Technology (ICT) school at the University of Tasmania (UTAS), is responsible for developing work-ready graduates for a very broad local ICT industry, and for providing research collaborations with industry and government. An external School review conducted in 2011 recommended the removal of the two existing undergraduate degrees (a Bachelor of Computing and a Bachelor of Information Systems) and the creation of a single bachelors degree. This degree would be the only undergraduate ICT degree within Tasmania.

To ensure that any one individual did not overly influence the new degree, the development was led by a working party consisting of eight academics heavily interested and experienced in teaching and learning with a variety of different characteristics and backgrounds. During the development phase the working party met on an almost weekly basis for approximately six months, with regular meetings with all School staff. It has been shown that involving academics that will be responsible for the implementation of a new curriculum during the design phase builds a sense of ownership that will facilitate change (Elizondo-Montemayor 2008).

To identify and incorporate crucial employability skills for graduates five forums were held with local and national ICT industry, government and pre-tertiary educators: three forums near the start of the process to identify career outcomes, and role specific and complementary knowledge; and two forums near the end of the process to receive feedback on the proposed course structure and curriculum. Over thirty different industry representatives participated, with some overlap between the people that attended the early and late forums. Organisations with varying numbers of ICT employees from one to thousands were represented. Most almost exclusively employed ICT graduates, with the local businesses predominantly hiring UTAS ICT graduates.

The development process was strongly guided by the seven-step process for curriculum design recommended by the Australian Computer Society (ACS 2011). Herbert, de Salas, et al (2013), reported on the methodology used to complete the first three steps — the identification of career outcomes, skill sets and the skill level of responsibility, including details on how feedback from each step was used to refine the list of career outcomes and skills. Herbert, Dermoudy, et al (2013) reported on the discussions held during the forums to complete steps four and five that identified the role specific and complementary knowledge that broadens graduate employability. This paper reports on the complexities of step six: “Design a course structure that incorporates ICT role specific knowledge with the core body of knowledge and other complementary knowledge as part of a holistic program of study” (ACS 2011).

As illustrated in Figure 1, ten months after commencing the curriculum development the final proposal was presented to an external academic review panel consisting of three leading academics in the field of ICT nationally, and along with cautionary advice relating to the implementation of the degree they concluded that it was a “very coherent, strong, contemporary” ICT curriculum (CIS 2013).

The new curriculum and course structure were developed to ensure:
• graduates can achieve career outcomes;
• inclusion of the body of knowledge for accreditation;
• a minimal number of units was required;
• adherence to course structure policy;
• increased commencement rates and decreased attrition rates of domestic and international students; and
• increased progression rates to research higher degrees.

This paper focuses on the influence of the above constraints on the final curriculum and course structure and how a balance between these at times conflicting constraints was achieved.
Graduates can achieve career outcomes

Graduate career prospects are one of the major factors influencing applicants when they select their course. Unfortunately, it is often unclear whether the careers were identified as part of the curriculum development process and there is little evidence that advertised career outcomes are really attainable by graduates. Calitz et al. (2011) stated “universities must link and publish computing programs, linking each program with specific career tracks, indicating specific career specialisation and knowledge”. To ensure that graduates can achieve the stated career outcomes it was necessary to identify the career outcomes for which there was demand by local and national industry, identify the combinations of skills required for the attainment of those career outcomes and finally, to ensure the curriculum included the knowledge requirements to enable each of the skills to be practised.

The fast-changing nature of technology has implications for ICT careers, as existing career titles and their attendant skill sets are disrupted, transformed or replaced (AWPA 2013). Our investigation, in 2012, indicated that there appeared to be no nationally recognised standard set of ICT career titles and definitions, despite many calls for this to be established (AGIMO 2007; Koppi & Naghdy 2009). An interactive ICT career streams diagram with 55 ICT careers (QLD Government 2013) was used to guide the identification of the degree’s career outcomes as it provided an objective externally-validated set of ICT career definitions rather than one developed subjectively by the academics.

18 members of the local and national ICT industry participated in an exercise to identify desirable career outcomes for the new degree. The participants identified 17 careers (out of the 55 on the diagram) that they had employed a graduate into in the past or would employ a graduate into in the future. The academic working party identified 24 careers; 15 of which were also identified by industry. A broad range of career outcomes was required in order to ensure the construction of a non-specialised ICT degree which would meet the demands of local and national industry, have wide appeal, and give graduates a range of world-wide career options. ICT Researcher, was not on the ICT career streams diagram (QLD Government 2013), however, given that a constraint was increased progression rates to a research degree it was included.

The Skills Framework for the Information Age (SFIA 2013) provides a comprehensive definition of a wide range of ICT skills practised by people working in ICT. Specifically, it lists 96 professional ICT skills, with each skill being mapped across seven levels of responsibility. The SFIA skill set was used to ensure that the course contained the skills that are expected in ICT graduates.

Using each identified potential career and the career streams diagram (QLD Government 2013), 38 of the 96 skills defined by SFIA were identified for the attainment of the career outcomes. Roberts et al. (2012) noted that providing students with an understanding of the social context in which society can benefit from ICT may be one of the most important changes to teaching that can be made for all students, so two additional complementary skills were included: HFIN (Human factors integration) and UNAN (Non-functional needs analysis).

Guided by the recommendation that undergraduate degrees should produce graduates with skills around SFIA level 4 of responsibility (ACS 2011), the identified skills were reviewed to determine whether each could be developed to the required level in the degree. There were some careers (Help Desk Operator, Technical Development Manager, and ICT Manager) that were initially selected that had skill levels that were too low or too high for an undergraduate degree.

Discussions were held at each forum to gather information about the role specific knowledge required to practise the skills and the complementary knowledge required to support the skill set or to broaden graduate employability. The following considerations to the curriculum design were raised:

- Similar to the findings of Pilgrim (2012), industry members believed it was essential that graduates be exposed to concepts in business analysis and process modelling, and project and change management.
- The issue of an increased use of off-shoring was identified as a possible impact on graduate software developer positions; this was mirrored in a report by NIEIR (2012). Software developer careers related to mobile application development were identified as becoming increasingly mainstream.
- The industry members were in favour of an “all-rounder” graduate. In accordance with the national findings (AWPA 2013), Tasmanian employers are more likely to choose a graduate with a broad range of ICT skills, with enough technical ICT content, as they have the ability to understand the needs of clients.
- The interviewees were insistent that the graduates are articulate professionals; there is no longer room in the industry for graduates who cannot relate well to business and clients. This supports the findings of Pilgrim (2012) that there are widespread views of “common deficiencies in the workplace readiness of new graduates particularly regarding the development of essential generic skills such as interpersonal and professional communications, business awareness and problem-solving abilities”.
- In Tasmania the demand for quality graduates is
currently exceeding domestic supply. Hence, as is the case nationally (AEI 2012), improving the quality — and in particular improving the quality of communication and interpersonal skills — of the international graduates was seen as a priority.

- Industry members wanted to participate more in the teaching program to bring in real-world examples and industry perspectives to the content. Koppi et al. (2010), reported that in responses to a recent student survey, respondents requested greater industry involvement in teaching with practical and relevant industry-based technologies and real examples.

Using insight developed from the collected data and the discussions a final set of career outcomes was identified that would guide the curriculum development. Categories were developed to distinguish the differences in the attainability of these career outcomes to assist potential applicants. These were:

- **graduate roles** — all skills are to be fully developed and the role is suitable for graduates;
- **career roles** — all theoretical skills are included and the role is suitable for graduates who have acquired a years experience and shown a detailed understanding of ICT and how it works within the business;
- **non-goal roles** — all the skills would be developed but the delivery of the content and discussion would not be focused towards these roles; and
- **partially qualified roles** — some key skills may be omitted or not developed to the required level.

30 career outcomes were identified and categorised for the curriculum, shown in Table 1. Table 2 lists the 37 SFIA skills from four SFIA categories (thirteen to level 4, twenty-four to level 5) that were identified. Herbert, de Salas, et al (2013) includes a different table relating the skills to career outcomes. Categories were developed to distinguish the differences in the attainability of these career outcomes to assist potential applicants. These were:

- **graduate roles** — all skills are to be fully developed and the role is suitable for graduates;
- **career roles** — all theoretical skills are included and the role is suitable for graduates who have acquired a years experience and shown a detailed understanding of ICT and how it works within the business;
- **non-goal roles** — all the skills would be developed but the delivery of the content and discussion would not be focused towards these roles; and
- **partially qualified roles** — some key skills may be omitted or not developed to the required level.

Table 2: SFIA skills and level of responsibility

with all 37 skills can be found at CIS (2013). Note Table 4 relates the unit code to unit title. Table 3 also demonstrates that to develop depth in a skill, the knowledge requirements have been embedded in units over the full three years of the course.

The final course structure includes elective units, which either expand on the material in a knowledge area covered in the core units, or introduce new knowledge areas to the curriculum as essential building blocks for some career outcomes, or are within the research directions of the School. Table 4 indicates the core and elective units that should be taken to qualify a student for a particular career outcome based on the skills required for that career, the complete table for all identified career outcomes can be found at CIS (2013).
Table 3: Extract of a table linking a SFIA code with the units that develop the skill

<table>
<thead>
<tr>
<th>SFIA Code</th>
<th>Introductory Units</th>
<th>Intermediate Units</th>
<th>Advanced Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUAN</td>
<td>KIT105</td>
<td>KIT203, KIT204</td>
<td>KIT301, KIT303</td>
</tr>
<tr>
<td>DTAN</td>
<td>KIT102</td>
<td>KIT204, KIT206</td>
<td>KIT301, KIT306</td>
</tr>
<tr>
<td>PROG</td>
<td>KIT101, KIT102, KIT103, KIT104, KIT107, KIT108, KIT109</td>
<td>KIT202, KIT205, KIT206, KIT207, KIT208, KIT212</td>
<td>KIT301, KIT302, KIT303, KIT307, KIT308, KIT309</td>
</tr>
<tr>
<td>ITOP</td>
<td>KIT102</td>
<td>KIT201</td>
<td>KIT304</td>
</tr>
<tr>
<td>HFIN, UNAN</td>
<td>KIT102, KIT105, KIT106, KIT109</td>
<td>KIT202, KIT206, KIT207, KIT208</td>
<td>KIT301, KIT302, KIT305, KIT311</td>
</tr>
<tr>
<td>SCAD, SCTY</td>
<td>KIT102, KIT104</td>
<td>KIT201, KIT202</td>
<td>KIT304</td>
</tr>
<tr>
<td>PRMG, CNSL</td>
<td>KIT105</td>
<td>KIT203, KIT204, KIT206</td>
<td>KIT301, KIT302, KIT303</td>
</tr>
</tbody>
</table>

Table 4: Units that develop the skills for a particular career outcome

<table>
<thead>
<tr>
<th>Core units in ICT Professional major</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT101</td>
</tr>
<tr>
<td>KIT103</td>
</tr>
<tr>
<td>KIT105</td>
</tr>
<tr>
<td>KIT106</td>
</tr>
<tr>
<td>KIT203</td>
</tr>
<tr>
<td>KIT204</td>
</tr>
<tr>
<td>KIT301</td>
</tr>
<tr>
<td>KIT302</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core units in Information Technology minor</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT102</td>
</tr>
<tr>
<td>KIT104</td>
</tr>
<tr>
<td>KIT201</td>
</tr>
<tr>
<td>KIT202</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core units in Software Development major and Games and Creative Technology major</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT107</td>
</tr>
<tr>
<td>KIT205</td>
</tr>
<tr>
<td>KIT305</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remaining core units in Software Development major</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT108</td>
</tr>
<tr>
<td>KIT206</td>
</tr>
<tr>
<td>KIT303</td>
</tr>
<tr>
<td>KIT304</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remaining core units in Games and Creative Technology major</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT109</td>
</tr>
<tr>
<td>KIT207</td>
</tr>
</tbody>
</table>

Restricted elective: KIT308, KIT309 or KIT311

<table>
<thead>
<tr>
<th>Remaining coursework elective units</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIT208</td>
</tr>
<tr>
<td>KIT212</td>
</tr>
<tr>
<td>KIT306</td>
</tr>
<tr>
<td>KIT308</td>
</tr>
<tr>
<td>KIT309</td>
</tr>
<tr>
<td>KIT311</td>
</tr>
</tbody>
</table>

Key

| Essential unit for this career | Recommended unit for this career | Contains relevant material for this career |
3 Includes the knowledge for accreditation

The Australian Computer Society (ACS) is our accrediting body; accreditation demonstrates to industry and applicants that a course contains the requirements of the ICT profession. To attain ACS accreditation a course must include the core body of knowledge (ACS 2011). Rather than using a curriculum-driven approach, where the degree is based purely on existing curricula that have been developed elsewhere, an approach was used which blended:

- Curriculum-driven — based on externally-endorsed curricula to ensure the inclusion of fundamental knowledge and informed by experts in ICT curricula design, rather than outspoken staff members (Gruba et al 2004) which results in an ‘individualistic’ curriculum (Henkel and Kogan 1999) with a loose coupling of units as well as a large number of units;
- Market-driven — based on career outcomes in demand by local and national industry members to ensure the inclusion of the required underlying knowledge. When emphasis is placed on employment objectives the resulting curricula are more directed and coherent (Henkel and Kogan 1999); and
- Discipline-driven — encompassing study across the boundaries of traditional disciplines to create a graduate who can become the “T-shaped” professional, described in AWPA (2013) as having broad knowledge and deep expertise, including technical skills, subject knowledge and soft skills (such as communication and business skills).

The ACS core body of knowledge (ACS 2011) is based on the Association for Computing Machinery (ACM 2013) curricula recommendations. The ACM provides curricula recommendations in five sub-disciplines: Computer Science, Computer Engineering, Information Systems, Information Technology, and Software Engineering.

Prior to commencing the identification of career outcomes it was considered important for all staff to renew their understanding of contemporary international curricula. This was partially to counter the issues raised by Gruba et al (2004) that outspoken individuals, rather than academic merit and external curricula predominantly drive curriculum change. The content recommendations for all ACM curricula were reviewed, as was the International Game Developers Association curriculum framework (IGDA 2008). These curricula were useful but the age of some diminished their utility. Curricula of national and international providers of ICT courses were also investigated to augment the curricula recommendations. In doing so, a portfolio of broad knowledge areas that a graduate would reasonably be expected to have was developed.

To ensure ACS accreditation the ACS core body of knowledge (ACS 2011) was included in the final curriculum. To ensure the curriculum included the knowledge requirements to enable each of the skills for each identified career outcome to be practised most topics from three ACM curricula were also included. All topics from the ACM IT core curriculum (ACM 2008) are included as are the majority of the ACM IT elective topics — either in the core or elective units of the degree. Everything from the core of the beta version of the ACM CS curriculum (ACM 2013) has been included in the core of the degree, with the exception of parallel and distributed computing which is included in two elective units. From the ACM IS core curriculum (ACM 2010) only IS strategy, management and acquisition, and enterprise architecture are not totally covered in the curriculum, but they are to be fully covered in an accompanying postgraduate coursework degree. From the ACM IS elective curriculum quite a large number of topics are covered as a result of the skills analysis for the career outcomes.

4 A minimal number of units

Due to a contracting staff profile, the cross campus nature of the School and significant pressure for increased research output an external School review conducted in 2011 recommended a reduction in undergraduate coursework unit offerings from 50 units to just 30 units.

Such pressure mandated that each unit maximized its contribution by:

- providing graduates with the essential technical and non-technical ICT skills and professional skills to enhance the Tasmanian ICT industry;
- inspiring students towards an ICT research career to increase the research output of the School; and/or
- attracting non-ICT students into the units by providing complementary knowledge to their chosen discipline.

Table 4 also shows the complete set of undergraduate coursework units. The equivalent of 28 coursework units (15 core units, and 13 elective units) were identified to cover the ACS core body of knowledge (ACS 2011), the knowledge requirements to develop the skills to the required level, and the core from the three ACM curricula (2013). To further reduce the load a few units will be offered on a two-yearly rotation basis and two units are being co-delivered by the School of Maths and Physics.

Every unit facilitates the development of a set of SFIA skills. An extract of this mapping is shown in Table 5; the full table with all units can be found at CIS (2013).

As a result of the curricula review the ACM IT curriculum (2008) was chosen as the basis for the new curriculum but it was resolved to include some aspects from the ACM IS (2010), ACM CS (2013) and IGDA (2008) curricula to encompass study across the boundaries of disciplines to ensure coverage of complementary knowledge to create a well-rounded graduate. For example Secure Web Programming works towards the partial development of nine SFIA skills; 22 hours of related material is included in the core of the ACM IT curriculum, yet none is in the ACM CS core curriculum. Another example is Data Structures and Algorithms that works towards the partial development of three SFIA skills; 28 hours of related material is included in the core of ACM CS curriculum but very little in the ACM IT curriculum. The final example is ICT Solutions Analysis for Business that works towards the partial development of sixteen SFIA skills; extensive material for this unit is extracted from the ACM IS curriculum but very little is in either the ACM IT curriculum or the ACM CS curriculum.
5 Adhere to course structure policy

The ACM IT curriculum (2008) recommends a four-year course. Discussions were held within the School and also with the industry representatives regarding the duration for the new degree. A four-year degree was not supported if the knowledge requirements could be covered in three years — partly because the graduates continue to learn and consolidate their skills during their employment. At UTAS there are only a few four-year undergraduate degrees and nationally there are very few four-year ICT degrees. To remain competitive with ICT degrees offered in other States and local degrees in other disciplines — which was essential for the local industry and the University — a three-year degree was developed. Qualified graduates that want to continue into a research career may append a one-year Honours degree and consolidation of their skills during their employment. At UTAS there are only a few four-year undergraduate degrees and nationally there are very few four-year ICT degrees. To remain competitive with ICT degrees offered in other States and local degrees in other disciplines — which was essential for the local industry and the University — a three-year degree was developed. Qualiﬁed graduates that want to continue into a research career may append a one-year Honours degree and consider being given to an alternative one-year Professional Honours for graduates that want to pursue further coursework and professional development. When asked, the stakeholders (academia, industry, and pre-tertiary educators) welcomed the recommendation of having a single degree with a reduced number of electives and a few distinct majors as this would remove confusion for applicants and would ensure that all ICT graduates had a balance of technical, non-technical and professional skills. UTAS (2013) has two course structure models summarised below. The industry representatives were polled and there was overwhelming support for the more intensive (Specialist) structure. The main argument for this was that it allowed coverage of enough technical material and more of the ACM curricula (2013) core content — particularly important given the decision to have a three-year degree rather than the recommended four-year degree — while also providing opportunity for the students to develop as professionals.

Table 5: Extract of a table illustrating which SFIA skills that will be developed within a unit

<table>
<thead>
<tr>
<th>Unit Title</th>
<th>SFIA code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Organisation and Visualisation</td>
<td>METL, PROG, DTAN, HFIN, UNAN, ICPM, SCAD, SCTY, INAN, TECH, DBDS, DBAD, IRMG, INCA</td>
</tr>
<tr>
<td>Computational Science</td>
<td>METL, PROG, TECH, RSCH</td>
</tr>
<tr>
<td>ICT Professional Practices</td>
<td>METL, BUAN, HFIN, UNAN, SCTY, CNSL, RLMT, ARCH</td>
</tr>
<tr>
<td>ICT Impact and Emerging Technology</td>
<td>RSCH, HFIN, UNAN, EMRG, ARCH</td>
</tr>
<tr>
<td>Secure Web Programming</td>
<td>METL, DTAN, PROG, HFIN, ICPM, SCAD, SCTY, TECH, INCA</td>
</tr>
<tr>
<td>ICT Solutions Analysis for Business</td>
<td>METL, BUAN, DTAN, PROF, UNAN, INAN, PRMG, CNSL, DESN, PBMG, BPRE, RLMT, BENM, IRMG, ARCH, DLGM</td>
</tr>
<tr>
<td>Data Structures and Algorithms</td>
<td>METL, PROG, TECH</td>
</tr>
<tr>
<td>Mobile Application Development</td>
<td>METL, PROG, ICPM, HFIN, UNAN, TECH, RSCH, EMRG, INCA</td>
</tr>
<tr>
<td>Data Analytics</td>
<td>METL, DTAN, RSCH, SCAD, INAN, TECH, DBDS, IRMG, INCA</td>
</tr>
</tbody>
</table>

Having identified a possible set of units it was necessary to identify what should be core units and what should be electives. It was also necessary to sequence the units into majors and minors. In the main, decisions about what should be core was determined by the requirements for ACS accreditation (2011). Using this constraint as a guide the units were grouped into possible majors and minors and then the remaining units for the majors were selected on the basis of either meeting the skill requirements for particular career outcomes or to ensure that the degree content was attractive to applicants. Table 4 also demonstrates how the units are divided into three majors and minor and electives. Within the Bachelor of Information and Communication Technology (BICT) at UTAS students will complete two majors and a minor:

- All students complete an “ICT Professional” major. This reversed major provides a breadth of professional skills such as teamwork, ethics, communication and interpersonal skills, entrepreneurship, and problem solving within the four introductory-level units. In response to the demands for business acumen in our graduates all students will be required to complete units in entrepreneurship, project management, requirements analysis, as well as business modelling and analysis at the intermediate-level. The major culminates in a capstone project at the advanced level to ensure these professional skills are reinforced throughout the course.

- All students also complete an “Information Technology” minor, which includes elements of the core ACM IT curriculum (2008) in the areas of Information Management, Networking, Web Systems and Technologies, Platform Technologies, and System Administration and Maintenance. The latter is also covered in depth in an accompanying major.

- Students will choose their second major based on their desired career outcomes and interest areas. The two majors: “Software Development”, or “Games and Creative Technology” have the same compulsory programming units at each level, including a mobile application development unit at the advanced level. Both majors have an ICT-restricted elective at the advanced level allowing some choice. The remaining units in the majors vary to give the students specific skills for their desired career paths. This structure allows students the flexibility to change their major easily should their interests change.

The ACM IT curriculum (2008) comes with two recommendations for presenting the core curriculum: Integration-first approach (an early integrated view of the
basic knowledge areas of the IT pillars) or the Pillars-first approach (introduce the detail of the IT pillars first, integration later). A Pillars-first approach was adopted as it was noted that this was better for allowing students to transfer into the degree from another course — most international students articulate into the intermediate-level. A noted disadvantage of the Pillars-first approach is that it does not provide an overview of how all the core material of an IT curriculum fits together and tends to present the details of each pillar in a more-isolated context. To overcome these disadvantages some of the features of the Integration-first approach were used:

- Breadth in ICT topics is introduced through the range of units provided at the introductory level. The degree is structured such that all eight first year units are pre-defined; the students primary opportunity for choice is that of their major. Depth has been created with units to be offered at all year-levels in a hierarchy requiring pre-requisites and with integrated content.
- There are two key pervasive topics to be developed throughout the degree: security and information assurance, and HCI and user-centredness. Both these elements are core in both the ACM IT (2008) and ACM CS (2013) curricula. As was stated in the ACM IT curriculum, these topics did not seem to belong in a single specific unit. As shown in Table 3, the SFIA skills HFIN (Human Factors Integration), UNAN (Usability Requirements Analysis), SCAD (Security Administration), and SCTY (Security Information) are developed across a number of units relating the skill to a wide-range of ICT application areas at all levels of the degree.
- Information such as that contained in the tables throughout this paper will be used to demonstrate to students how the curriculum content relates to each career outcome and also how skill development towards a career is integrated across a number of units. This information will be made available within the core unit ICT Professional Practices which is scheduled for their first semester so it will also be timely help for students to identify the relevant major and elective units for their chosen careers.

6 Attractive to domestic and international applicants

ICT degrees continue to rank poorly on the list of preferred courses for students applying for university places. In 2013, the broad field of IT ranked second lowest of highest preference applications out of ten fields of education (DIISRTE 2013). In 2012, the ACS released figures (ACS 2012) indicating that the number of domestic students graduating from ICT courses had halved over the last decade; down from 9093 in 2003 to an expected 4547 in 2013. Of those who commence only 54.6% complete their ICT course. And yet, since 2003 there has been a 31% growth in ICT industry employment. This indicates that there is a huge disconnection between supply and demand; this shortfall between people starting and finishing an ICT course poses a major risk to the ICT sector and even the national economy (ACS 2012).

There is a general consensus that ICT has an image problem at a time when the need for skilled ICT professionals has never been greater, and that there is a lack of awareness of the wide range of career possibilities in ICT (AWPA 2013). A career in ICT is perceived as male-dominated, repetitive, isolated, and focused on the technical rather than the professional (AWPA 2013). To counter this negative and inaccurate perception, and to promote the future growth of the industry, it is essential that the career outcomes for modern ICT curricula reflect the ever-expanding reality of ICT careers now available (ACM IT 2008) and the curriculum is designed such that graduates can attain these careers (von Konkoly 2008).

To meet the demand for ICT graduates the advertised curriculum has to spark enough interest in a range of potential applicants, not just those currently studying ICT related subjects at pre-tertiary level. Also with such a reduced set of units, each unit has to be attractive, and not just to students completing the BICT degree but also as electives to students within other degrees (e.g. business, science, and arts degrees where students can take eight elective units). Alongside identifying a wide range of career outcomes and ensuring the necessary skills were developed a number of measures were taken to increase the attractiveness of the degree and the units in order to increase commencement rates and decrease attrition rates:

- ICT courses are seen to lack a workplace or business focus and to lack practical application leading to high attrition rates (Roberts et al 2012). By incorporating business and professional skills the course should be more attractive to applicants through balancing the technical and non-technical focus and also through creating a balance between the application and theory (Roberts et al 2012) and will also deliver stronger graduates to the industry (AGIMO 2013).
- Pollitzer (2012) recommended including information and knowledge about the impact that ICT has on society to excite students about an ICT career. Consequently, a unit on the Impact of ICT and Emerging Technology has been created at the introductory level.
- An introductory Artificial Intelligence unit will be developed to attract and intrigue students from a range of disciplines (e.g. science, psychology, life science, fine art, etc) by research material from the field.
- An important predictor of attrition is previous ICT experience (Roberts et al 2012). The compulsory first-year programming unit will have a pre-requisite — hence ensuring that all students have some programming experience. The requirement for pre-requisite knowledge will facilitate a more interesting and challenging unit — which should have a positive impact on the retention of students with prior programming knowledge. Alternative pathways will be available to qualify students that do not have prior programming experience. For example, students could take an additional programming unit in their first year that provides the opportunity to develop the foundational skills, and confidence, to be successful. This approach has been shown to particularly address the attrition of female students as they are less likely to have the prior experience and a stronger foundation leads to higher marks thus increasing the satisfaction for female students (Roberts et al 2012).
To attract increased applications for ICT-related courses, many universities create programs to allow students to combine subjects from a number of disciplines and enabling students to extend their studies into ICT application areas (AGIMO 2013). Related courses that don’t impact on the available resources (e.g. staffing, number of units) that will attract additional students to the units have been created. Six four-year combined degrees have been created allowing students to complete 16 units in ICT and 16 units in another discipline. In particular, a combined degree with the Bachelor of Visual Communication has been created to attract the large number of students doing computer graphics at pre-tertiary level. This particular degree is designed for students aiming for a career in Graphic Design which industry members indicated were in high demand (Herbert, de Salas, et al 2013). A Computer Science major within the Bachelor of Science degree and an ICT major within the Bachelor of Business will also allow students that don’t want to do an entire degree in ICT to develop a subset of ICT skills.

Pollitzer (2012) recommended giving priority not only to the workforce-related skills but also to ICT skills needed for entrepreneurship and creativity in order to attract more applications for ICT courses. Entrepreneurship has been embedded as a theme throughout the ICT Professional major and creativity is a stream within the Games and Creative Technology major. At the forums, very few industry members identified a demand for the Game Developer career, but they did recognise that content related to games was a strong draw-card for applicants and they welcomed the potential increase in graduate numbers its inclusion could provide.

As indicated by the attrition rates the way in which ICT is taught clearly requires urgent consideration — particularly to reduce the attrition in female students (Roberts et al 2012). As recommended by Roberts et al (2012) there will be an increased use of small group class activities, which provide students with opportunities to undertake more active learning and increased interaction with other students and staff, allowing students to feel that they are active participants in their own learning.

International student commencements in Australia in university-level ICT courses over the last five years have consistently been at least 20% higher than domestic enrolments, and in contrast to only 54.6% of domestic students that complete, international students had a completion rate in 2012 of 85.8% per cent (ACS 2012). At UTAS in 2010 74% of commencing ICT students were international students (many studying offshore in China), whereas across all disciplines only 24% were international students, and, 22% of the international students enrolled were enrolled in an ICT degree (ACS 2012). With the School and the University dependent on these continued international enrolments it was essential that consideration be given to the extra features desired in an ICT degree by international students. With these students choosing to study in Australia making substantial investments in their future, there is scope for improvement to help these students to gain the tools to find appropriate jobs once they graduate (AEI 2012). A number of measures were taken to make the degree attractive to international students other than developing a degree that would receive accreditation (a key immigration factor for students):

• According to an AEI employer survey (AEI 2012) employers want more emphasis on developing communication skills and English-language skills among international students. Many international students are given advanced standing on the basis of prior learning in their country of origin. This often results in them being ‘slotted’ into second year and by-passing units that provide explicit and/or incidental induction to the institution, and which develop communication and teamwork skills. As a result many are technically competent but are not best able to compete for employment on graduation. The initial experiences of international students are extremely important, laying the foundation for their success in Australia (AEI 2012). A bridging unit, just for articulating students, has been created to redress these issues. This unit will contain all the core induction material and allow international students to develop the introductory-level teamwork and communication skills. Throughout the ICT Professional major units will focus on providing opportunities for interaction and engagement via teamwork and problem-based learning activities, to help all students to develop the communication and language skills desired by industry.

• An AEI survey of employers conducted in 2010 found that providing practical work experience was one of the main areas requiring more emphasis in an Australian education for international graduates (AEI 2012). Another AEI survey of Australia-educated international graduates found that the most commonly perceived barrier for graduates who had been unable to find work was a lack of work experience (AEI 2012). An elective industry placement unit has been created and is available to all students to gain genuine work experience in their second year in addition to the core capstone project experience that all students receive in their final year. Pilgrim (2011) noted the importance of introducing work-integrated learning early in the curriculum, rather than just relying upon the capstone units at the end. Roberts et al (2012) identified that team-based industry projects and work placements enable students to gain professional skills and strengthen their sense of the relevance of their ICT course while also ensuring that the curriculum is aligned with industry needs.

7 Increase progression to research degrees
Like many Australian universities, UTAS is re-positioning itself and seeking to increase its research reputation. The new curriculum needs to reflect the research directions of the School and it has to inspire more graduates to continue onto a research higher degree to increase the research output of the School.

The “teaching-research nexus” has been described in a variety of ways in the literature. Perhaps the most widely cited is Healey’s (2005) categorisation that proposes students experience research in four main ways:
• research-led — in which students learn about research findings;
• research-oriented — in which students learn about research processes;
• research-based — in which students learn as researchers; and
• research-tutored — in which students learn in small group discussions about research findings.

Within the new curriculum Artificial Intelligence, a key research field of the School, will be offered as a research-led introductory unit. Data Analytics is another research-led elective unit at the advanced level in which students will gain an understanding of the major research issues in the area of ‘big data’.

Koppi and Naghdy (2009) introduced the concept of the teaching-research-industry-learning (TRIL) nexus in ICT education. They highlighted that in addition to the relationships between teaching and research, industry is often at the forefront of ICT research and that understanding and strengthening the TRIL nexus would lead to curriculum improvements. Within the new curriculum there is an intention to embed research and industry guest speakers throughout all units to relate the content of each unit to research that is happening in the field and to what the students will experience in employment. Increased use of case-based teaching and learning will tie ICT content to application, enabling students to understand the context in which their knowledge will be applied (Roberts et al 2012).

Strazdins (2007) described a comprehensive effort to introduce a research culture throughout an entire undergraduate Computer Science curriculum. He notes, however, that a research-based ICT education may not be suitable for all students — especially those of lower ability and at lower undergraduate levels. Within the new curriculum there will be a steady increase from problem-based approaches in introductory and intermediate units to research-based approaches in the advanced level units.

Strazdins (2007) also comments on the view of some academics that it can be hard and time-consuming to include research-based assignments and group work when classes are large and teaching loads are high. In the new curriculum the number of coursework units has been significantly reduced decreasing the teaching load for staff. There will be a number of elective research-based units in the new curriculum that will have pre-requisites that restrict enrolment to the top students. Small class sizes and special experiences for these top students will hopefully inspire in them a desire to stay to complete higher degrees and pursue a career in research. ICT R&D project units have been introduced at all three levels of the degree for top students. As well as being research-based these units are also research-oriented as students will learn about research processes while conducting a research project. Finally, three advanced level ‘elite’ research-based and research-tutored units will be offered annually in which students will learn in small classes about research findings that relate to research focuses of the School delivered by research-intensive academics. For example, in 2014 elite units in eHealth, sensor networks, and artificial intelligence will be offered; each a significant research focus area of the School.

8 Conclusion

Curriculum design is a complex process that must be informed by stakeholders and developed from multiple perspectives. This paper reports on the development of an ICT curriculum that involved balancing a number of constraints, which, in the words of an external academic advisory panel, resulted in a “very coherent, strong, contemporary” ICT curriculum (CIS 2013).

A broad range of career outcomes and skill sets were identified by a balanced view of academic insight and industry demands — both being further supported by externally validated and industry-standard definitions. Decisions about what to include in the curriculum were guided by these career outcomes and skill sets and the curriculum was aligned with ACM international curricula (ACM 2013) and the ACS core body of knowledge (ACS 2011). The curriculum includes features that will be attractive to domestic and international applicants alike, and will promote progression to a research higher degree.

The pervasive themes (security and information assurance, and HCI and user-centredness) alongside the depth areas (professionalism, software development, games and creative technology, information technology) and the fact that development of each skill has been integrated across a wide-range of ICT topic areas and that the curriculum encompasses study across the boundaries of disciplines will generate well-rounded graduates who have a much better understanding of the relationship between the ICT content.

In focusing on a set of career goals, a course structure was designed that required a small number of units which consequently yielded a reduced teaching load for staff, and hence the creation of increased time for research. Although the number of units is relatively small they contribute to a broad range of career outcomes allowing enough choice for applicants to pursue their individual interests. Students can be assured that the advertised career outcomes are genuinely attainable and that the degree was developed with these career outcomes in mind. Furthermore, employers of the graduates can have confidence that the course contains the relevant skills that are expected of graduates.

9 References


Student and staff expectations of NZQA levels

Mike Lopez
Christchurch Polytechnic Institute of Technology
130 Madras Street
Christchurch, New Zealand
mike.lopez@cpit.ac.nz

Dobrila Lopez
Christchurch Polytechnic Institute of Technology
130 Madras Street
Christchurch, New Zealand
dobrila.lopez@cpit.ac.nz

Abstract
In this cross-sectional study, we surveyed 275 students from representative courses across four levels of the New Zealand Qualifications Framework (NZQF) to elicit their expectations of the nature of the skills and knowledge, and the degree of self-management and collaboration, required for their courses. We compared their responses to the expectations set out in the qualifications framework. We also surveyed 9 lecturers and compared their expectations to both students and the framework. We found significant differences between student and framework expectations; student expectations were clearly lower than the framework. Moreover, student expectations remained at a low level, even in higher level courses and the gap was wider at the higher levels. We also found significant differences between student and lecturer expectations. At the higher course levels, lecturer expectations were broadly between those of students and the framework, which suggests that lecturer expectations are a compromise between both of these. Any misalignment of expectations poses a challenge for educators. We suggest practical measures for aligning these expectations.

Keywords: Student expectations, lecturer expectations, accreditation expectations.

1 Introduction
The premise underpinning this study is that learning happens best when there is a close alignment between the expectations of students, educators and other stakeholders. In New Zealand, accreditation of University courses is overseen by the Academic Quality Agency (AQA). Courses from all other providers come under the auspices of the New Zealand Qualifications Authority (NZQA) Zealand. The NZQA publishes the New Zealand Qualifications Framework (NZQF), which is a comprehensive list of all quality assured qualifications in New Zealand (New Zealand Qualifications Authority, 2011). Similar frameworks exist in Australia (Australian Qualifications Council, 2013) and the UK (The Quality Assurance Agency for Higher Education, 2008).

The New Zealand framework is organized into ten levels. Broadly speaking, levels one to three map to school years. For example, New Zealand's National Certificates of Educational Achievement (NCEA) are national qualifications for senior secondary school students at levels one to three. Levels four to ten typically relate to post-secondary qualifications. Levels four to six are usually certificates and diplomas, although it is also possible to have these at all levels from one to eight. Undergraduate degrees are at level seven. Post-graduate certificates and diplomas are at level eight, a master’s degree is at level nine, and a doctoral degree is at level ten.

For each of the levels, the framework sets out clear expectations of the nature of skills, knowledge, student self-management, and how a student should interact with others. Given the central role of the framework in accreditation, one might expect to find a close alignment between the expectations of NZQA and those of educators and students. However, we had reason to believe some differences might be found. For example, Nunn and colleagues investigated student perceptions of desirable graduate characteristics (Nunn, et al., 1995) and found considerable differences from academic and employer characteristics.

Consequently, it seemed reasonable to expect that we too might find some differences.

Any misalignment poses a challenge for educators. With the on-going consumerisation of education, students are seen more and more as consumers of a service (Naidoo & Jamieson, 2005) – as customers. Somehow, educators need to achieve the stated educational goals while also meeting student expectations.

This study builds on a pilot study (Lopez, et al., 2013), which used a convenience sample and served, primarily, to validate the instrument. In contrast, the present study used a larger systematic sample which was more representative of the overall student body. The study aims to answer the question: How closely aligned are the expectations of students, lecturers and the NZQA?

The remainder of this paper is organized as follows. In section two, we discuss related work in the literature. In section three, we describe our approach to the study and our methodology. We present our findings in section four. We discuss the implications of our findings for teaching in section five. Finally, we discuss the limitations of the approach and identify areas where further work is required in section six.

2 Related Work
In the literature, a number of researchers have used external frameworks to analyse courses. In computing, several researchers have investigated the mapping of courses to Bloom’s taxonomy (Bloom, 1956). Bloom’s taxonomy is widely used in educational contexts to give an approximate indication of the cognitive depth needed for a task. Sanders and Mueller (2000) argued that courses in the early stages of a degree should be targeted at the lower Bloom’s levels, whereas later courses should be targeted at
the higher levels. Lister used the taxonomy to formulate course objectives for a sequence of programming courses (Lister, 2001). Howard and colleagues carried out a lesson-by-lesson analysis of depth in a CS2 course (Howard & Carver, 1996). Oliver and associates (2004) carried out a lecturer evaluation of the cognitive difficulty of a number of computing courses. Most of this work is grounded in the programming area, and underpinning most of this work is the assumption that in teaching programming “we have traditionally focused on the higher levels of the taxonomy and ignored the lower levels" (Lister & Leaney, 2003, p. 147).

Another widely used framework is the SOLO taxonomy (Biggs & Collis, 1982). In particular, SOLO has been used to map the cognitive complexity in programming. For example, Brabrand and Dahl (2007; 2009) used SOLO to analyse over 5000 intended learning outcomes, comparing those in Computer Science to those in Mathematics and natural science. Thompson (2007) used SOLO to develop assessment criteria for programming assignments. Sheard and colleagues (2008) used SOLO to explore the programming knowledge of novices. Lister and associates (2006) used SOLO to describe differences in the way students and educators solve small code reading exercises. Whalley and colleagues (2006) used SOLO and Bloom’s taxonomies to develop a question set for novice programmers. There are other less widely used taxonomies. For example, Fuller and associates carried out a literature review of the use of Bloom’s and SOLO taxonomies and proposed a two-dimensional matrix taxonomy (Fuller, et al., 2007).

Both SOLO and Bloom’s taxonomies have been widely used as a conceptual framework to analyse cognitive complexity in computer science. However, all of the studies cited represent an educator’s perspective, rather than that of a student.

To elicit a student perspective, we have to turn to the general tertiary education literature. However, research on student expectations is still sparse within this literature. Ldowden and colleagues (2011) investigated employer perceptions of the employability of new graduates and Weligamage and Sienthai (2003) compared student and employer perceptions. Round (2005) investigated broad student expectations of University in the context of understanding and enhancing student retention.

Despite these few examples, we found that, overall, student and lecturer expectations of course levels remain underexplored in the tertiary education literature. Despite the stated aim that “This assists learners when making decisions on which qualifications to undertake, and when, and where” (New Zealand Qualifications Authority, 2011, p. 3), the framework descriptors seem to be used more for communication between providers and the accreditation authority than for communication with students. Indeed Kemmis and associates note:

Student expectations and the broader set of expectations that flavour VET and HE are often quite different and are often implicitly embedded in subjects and courses. The process of making these differences explicit is left to the student making the transition. (Kemmis, et al., 2010, p. 30)

We believe that it is important that expectations are shared between students and educators, and not just left to the student. Our study aims to identify the extent to which student and lecturer expectations are aligned to each other and to the level descriptors in the framework.

3 Approach and method

In this cross sectional study, we used an anonymous questionnaire to survey lecturers and students. All lecturers in the authors’ department were invited to participate. Lecturers were supplied with participant information sheets and paper questionnaires and were invited to participate by email. Participation was voluntary and questionnaires were anonymous and without any demographic information.

For students, we used a systematic sampling frame. Representative courses were identified, in consultation with the respective programme leaders, from the level four Certificate in Information and Computing Technologies (CICIT) and from each semester of the Bachelor of Information and Computing Technologies (BICT), which has courses at levels five, six and seven. Students were recruited in these selected courses with permission of the respective lecturers. The students were supplied with participant information sheets and paper questionnaires. Participation was voluntary and the questionnaires were anonymous and without any demographic information.

3.1 Instrument

We used a custom questionnaire for the survey. The NZQA level of the course was known for students because separate batches of questionnaires were used for each course. We asked lecturers to respond to the same questions for a number of programmes and levels. For all participants, we used four questions to investigate characteristics of the levels relating to self-management, working in groups, skills and knowledge. We took the wording for the questions from the level descriptors in the NZQA accreditation document (New Zealand Qualifications Authority, 2011). To align our questions with the wording used in the NZQA document, we prefaced each student question with the stem: In this course, it is reasonable to expect a student to … For the lecturer questions, we asked lecturers to answer the question for a number of programmes and levels and used the stem: For a course in this program/level, it is reasonable to expect a student to …

We then presented the participant with a list of the exact wording used in the NZQA document to characterise the levels and asked the participants to indicate which they felt was closest to their expectation. As an example, Question2 in the student questionnaire is shown in Figure 1.

For this course, it is reasonable to expect that a student will:
• Not interact with others – students should work independently.
• Interact with others
• Collaborate with others
• Contribute to group performance and adapt own behaviour when interacting with others
• … and so on

Figure 1: Sample question
Note that, for completeness, we added the first of these options (not interact with others) to the framework. The framework starts with level 1 (interact with others).

3.2 Sample
The questionnaire was administered to 275 students and nine lecturers. All lecturers in the authors’ department were invited to participate. Of 32 possible lecturers, nine (28%) chose to participate. For the student sample, students enrolled in the chosen representative courses for each semester of study were invited to participate. These courses were either a compulsory course, or one that was deemed by the appropriate programme leader to be usually taken by those in the target cohort of students. Of 459 possible students, 275 (60%) chose to participate. The sample characteristics are summarised in Table 1.

Table 1: Sample characteristics

<table>
<thead>
<tr>
<th>Level</th>
<th>N</th>
<th>Response rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>59</td>
<td>71%</td>
</tr>
<tr>
<td>5</td>
<td>63</td>
<td>61%</td>
</tr>
<tr>
<td>6</td>
<td>85</td>
<td>51%</td>
</tr>
<tr>
<td>7</td>
<td>68</td>
<td>65%</td>
</tr>
<tr>
<td>All students</td>
<td>275</td>
<td>60%</td>
</tr>
<tr>
<td>All lecturers</td>
<td>9</td>
<td>28%</td>
</tr>
</tbody>
</table>

Because of the low number of lecturer participants, it is unlikely that the views captured are fully representative of the department. Nevertheless, the effect sizes found in those sampled were large enough to produce statistically significant results.

In summary, the NZQA level was known from the course for students and was captured explicitly from lecturers. Participants’ expectations were identified by matching the wording they chose to the NZQA framework.

3.2.1 Analysis
For descriptive statistics, we use the mode and mean. For statistical inference, we were interested in the question: how likely are these data if we are sampling randomly from a population with a mean of the expected NZQF level? Thus, our data are slightly unusual inasmuch as the population mean is known a priori. However, the variance is still estimated from the sample. To accommodate this, when comparing to the framework levels, we base inference on the standard error of the mean (SEM) and use a z-test for inference rather than the usual t-test. On the other hand, we used t-tests to compare student and lecturer expectations since the means of both of these were estimated from the sample.

4 Results
Expectations are clearly defined by the qualifications framework. Consequently, one might expect the expectations of students and lecturers to be closely aligned with the framework and, thus, that the expected level description from the framework would be chosen in most cases. However, this was not the case. We begin this section with student expectations and then present lecturer expectations. Table 2 shows the proportion of students who chose the expected level according to the framework.

As can be seen, only 10% of the overall student choices were at the expected level, and a strong preference for the expected level is only visible for question one among the level four cohort of students. This poses a major challenge for educators. A course taught at the expected framework level will fail to meet the expectations of the majority, on average about 90%, of the students. Consequently, from an educational perspective, it is natural to ask whether a lecturer could start with expectations at a lower level and then work with students to build progressively to the framework level. However, this would still fail to meet expectations. Table 3 shows the modal levels expectations chosen by students in each course level, together with the proportion who made that choice.

Note that, for completeness, we added the first of these options (not interact with others) to the framework. Consequently, one might expect the expected level description from the framework would be chosen in most cases. However, this was not the case. We begin this section with student expectations and then present lecturer expectations. Table 2 shows the proportion of students who chose the expected level according to the framework.

Note: Rows show the level of the course in which the student was enrolled. Each cell shows the percentage of students choosing the expected NZQA framework level. Q1 is self-management, Q2 is collaboration, Q3 is knowledge and Q4 is skills.

Table 2: Proportion choosing the expected level

<table>
<thead>
<tr>
<th>Level</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>36%</td>
<td>7%</td>
<td>5%</td>
<td>15%</td>
<td>16%</td>
</tr>
<tr>
<td>5</td>
<td>8%</td>
<td>10%</td>
<td>13%</td>
<td>22%</td>
<td>13%</td>
</tr>
<tr>
<td>6</td>
<td>1%</td>
<td>4%</td>
<td>6%</td>
<td>26%</td>
<td>9%</td>
</tr>
<tr>
<td>7</td>
<td>0%</td>
<td>0%</td>
<td>15%</td>
<td>6%</td>
<td>5%</td>
</tr>
<tr>
<td>Overall</td>
<td>10%</td>
<td>5%</td>
<td>9%</td>
<td>18%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Note: Rows show the level of the course in which the student was enrolled. Each cell shows the percentage of students choosing the expected NZQA framework level. Q1 is self-management, Q2 is collaboration, Q3 is knowledge and Q4 is skills.

Table 3: Modal levels chosen by students

<table>
<thead>
<tr>
<th>Level</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4 (36%)</td>
<td>3 (54%)</td>
<td>2 (17%)</td>
<td>5 (29%)</td>
</tr>
<tr>
<td>5</td>
<td>4 (38%)</td>
<td>3 (37%)</td>
<td>3 (29%)</td>
<td>4 (25%)</td>
</tr>
<tr>
<td>6</td>
<td>4 (36%)</td>
<td>3 (49%)</td>
<td>3 (24%)</td>
<td>4 (31%)</td>
</tr>
<tr>
<td>7</td>
<td>4 (31%)</td>
<td>3 (40%)</td>
<td>4 (16%)</td>
<td>5 (28%)</td>
</tr>
<tr>
<td>(all)</td>
<td>4 (35%)</td>
<td>3 (44%)</td>
<td>3 (21%)</td>
<td>5 (24%)</td>
</tr>
</tbody>
</table>

Note: Rows show the NZQA level of the course in which the student was enrolled. Each cell shows the level of the most popular response, followed by the percentage of students who chose that response.

It can be seen that the percentage is above 50% in only one case: question 2 (collaboration) for the cohort of students at level four. In all other cases, there is no single level expectation that could be chosen which would satisfy the majority of the class. For example, if one teaches a level six course with the modal level 3 expectation of collaboration, this will still fail to meet the expectations of the majority of the class. Moreover, comparison across levels suggests that one could teach courses at all of the levels from four to seven with the same approaches to self-management, collaboration, knowledge and skills without affecting student expectations much. The last row in this table, which is a composite across levels, supports this observation; the percentages are not very different from those in the rest of the table.

The following sections show student and lecturer expectations for each of the individual questions.
4.1 Question One: Self-Management

This question asked about the degree of self-management that a student could be expected to show. Table 4 shows the mean expectation of students by cohort and level.

Table 4: Student expectations of self-management

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.00</td>
<td>0.17</td>
<td>n.s.</td>
</tr>
<tr>
<td>5</td>
<td>3.59</td>
<td>0.15</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>6</td>
<td>3.29</td>
<td>0.10</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>7</td>
<td>3.50</td>
<td>0.15</td>
<td>&lt; .001</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

As can be seen, the mean expectation in level four courses is consistent with the framework expectation. However, expectations at levels five, six, and seven are not only below those of the framework, but are also significantly below level four expectations. Moreover, there is an apparent fall in student expectations as they progress through the levels with their study. Overall, the mean student expectation can be characterised as between level three (requiring major responsibility for own learning and performance) and level four (self-management of learning and performance under broad guidance).

Lecturer expectations of self-management are shown in Table 5.

Table 5: Lecturer expectations of self-management

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.78</td>
<td>0.43</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>5</td>
<td>3.07</td>
<td>0.32</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>6</td>
<td>4.00</td>
<td>0.23</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>7</td>
<td>4.83</td>
<td>0.40</td>
<td>&lt; .001</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

From this table, it can be seen that lecturer expectations are significantly below the NZQF framework expectations at all levels. However, there is a clear increasing pattern of expectations and the data suggest that lecturer expectations are lower than those of students at level four, but become higher at levels six and seven. Individual t-tests confirm these differences between student and lecturer expectations. At level four, the mean lecturers’ expectation of 1.78 is significantly lower than the student expectation of 4.00 (p=.041). At level five, the mean lecturers’ expectation of 3.07 is not significantly different from the students’ expectation of 3.59 (p=.151). At level six, the mean lecturers’ expectation of 4.00 is significantly higher than the mean of 3.29 for students (p=.013). At level seven, the mean lecturer expectation of 4.83 was significantly higher than the mean of 3.50 for students (p=.019).

A summary plot of student and lecturer expectations of self-management is shown in Figure 2.

4.2 Question Two: Collaboration

This question related to the degree to which a student could be expected to collaborate with others. Table 6 shows the mean level of student expectation by cohort.

Table 6: Student expectations of collaboration

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.15</td>
<td>0.15</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>5</td>
<td>2.85</td>
<td>0.17</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>6</td>
<td>2.58</td>
<td>0.13</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>7</td>
<td>2.75</td>
<td>0.19</td>
<td>&lt; .001</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

As can be seen, the mean student expectation is significantly below the framework expectation at all levels. Moreover, expectations at levels five, six, and seven are not only below those of the framework, but are also significantly below level four expectations. There is a clear falling pattern of expectations as students carry on with their study to higher levels. Overall, mean student expectations can be characterised as between level two (collaborate with others) and level three (contribute to group performance and adapt own behaviour when interacting with others). The expectation is significantly below level four (demonstrate some responsibility for the performance of others) at all levels.

Lecturer expectations of collaboration are shown in Table 7. As can be seen, the mean expectation is significantly below the framework expectation at all levels. There is a clear rising pattern across the levels, although the low rate of increase means that the gap between lecturer and framework expectations increases at the higher levels.
Table 7: lecturer expectations of collaboration

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.11</td>
<td>0.45</td>
<td>&lt;.001</td>
</tr>
<tr>
<td>5</td>
<td>3.00</td>
<td>0.33</td>
<td>&lt;.001</td>
</tr>
<tr>
<td>6</td>
<td>3.77</td>
<td>0.34</td>
<td>&lt;.001</td>
</tr>
<tr>
<td>7</td>
<td>5.17</td>
<td>0.40</td>
<td>&lt;.001</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

There are also differences between student and lecturer expectations. At levels four and five, there is no significant difference between students’ and lecturers’ expectations; significance is p=.056 and p=.699 respectively. At level six, the mean lecturer expectation of 3.77 is significantly higher than the mean of 2.58 for students (p=0.005). At level seven, the mean lecturer expectation of 5.17 is significantly higher than the mean of 2.75 for students (p<.001).

A plot of both student and lecturer expectations is shown in Figure 3.

Figure 3: Collaboration

It can be seen that mean student expectations are in line with the framework at level four, but are significantly below the framework at all higher levels. Moreover, the difference between these expectations and those of the framework increases as students progress through the levels, ending about two levels below the framework expectation at level seven.

Overall, the mean expectation can be characterised as being between level 3 (some operational and theoretical knowledge in a field of work or study) and level 4 (broad operational and theoretical knowledge in a field of work or study) for level four to six cohorts and level 5 (broad operational or technical and theoretical knowledge within a specific field of work or study) for the level seven cohort. In particular, we note that both level six and level seven student samples had mean expectations well below the NQF expectation for level 6 (specialised technical or theoretical knowledge with depth in a field of work or study).

The corresponding lecturer expectations of knowledge are shown in Table 9.

Table 9: Lecturer expectations of knowledge

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.22</td>
<td>0.49</td>
<td>&lt;.001</td>
</tr>
<tr>
<td>5</td>
<td>3.69</td>
<td>0.34</td>
<td>&lt;.001</td>
</tr>
<tr>
<td>6</td>
<td>4.80</td>
<td>0.37</td>
<td>.001</td>
</tr>
<tr>
<td>7</td>
<td>6.86</td>
<td>0.26</td>
<td>n.s. (.584)</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

As can be seen, the mean expectation is significantly below the framework expectation at levels four, five and six, but not significantly lower at level seven. Although lecturer expectations start well below those of the framework, there a clear rising pattern is visible with the difference from framework expectations reducing progressively and ending in alignment with the framework at level seven. Lecturer expectations are significantly below those of students at level four (p=.038), in line with student expectations at level five (p=.498, n.s.) and significantly above student expectations at level 6 (p=.017) and level seven (p < .001). A plot of student and lecturer expectations of knowledge is shown in Figure 4.
It can be seen that students start with expectations in line with the expectations of the framework but these fall progressively below the framework at subsequent levels. There is a visible rise at level seven, but the increase is only to about level five: two levels below the framework expectation. In contrast, lecturer expectations start about two levels below the framework and rise progressively achieving alignment at level seven.

4.4 Question four: Skills
The fourth question was about the nature of skills. Table 10 shows the student expectations by level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.48</td>
<td>0.29</td>
<td>n.s. (p=.094)</td>
</tr>
<tr>
<td>5</td>
<td>5.10</td>
<td>0.22</td>
<td>n.s. (p=.659)</td>
</tr>
<tr>
<td>6</td>
<td>4.88</td>
<td>0.17</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>7</td>
<td>5.24</td>
<td>0.24</td>
<td>&lt; .001</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

Overall, the mean expectation of the level four and five cohorts was consistent with the framework expectations of “select and apply solutions to familiar and sometimes unfamiliar problems” and “select and apply a range of solutions to familiar and sometimes unfamiliar problems”, respectively. However, these expectations do not appear to rise above level five; the mean expectation of the level six and level seven cohorts is consistent with level five. In particular, we note that, the mean expectations at both levels six and seven were both significantly below the framework expectation at level six (analyse and generate solutions to familiar and unfamiliar problems). Consequently, the gap between student expectations and the framework widens at the higher levels, falling progressively short of the framework expectations.

Lecturer expectations of skills are shown in Table 11.

<table>
<thead>
<tr>
<th>Level</th>
<th>Mean</th>
<th>Std. Err</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.44</td>
<td>0.34</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>5</td>
<td>3.69</td>
<td>0.28</td>
<td>&lt; .001</td>
</tr>
<tr>
<td>6</td>
<td>5.07</td>
<td>0.30</td>
<td>.002</td>
</tr>
<tr>
<td>7</td>
<td>7.00</td>
<td>0.31</td>
<td>n.s. (1.0)</td>
</tr>
</tbody>
</table>

Note: Sig is the probability of observing a mean this low in a random sample drawn from a population with the expected mean.

As can be seen, lecturer expectations start out significantly below the framework and remain significantly below the framework expectations until level seven. However, there is a clear rising pattern and the difference from the framework progressively narrows, reaching alignment at level seven.

Lecturer expectations are significantly lower than those of students at level four (p < .001) and level five (p < .001). They are in alignment at level six (p=.593, n.s.) and significantly higher at level seven (p < .001).

A plot comparing student and lecturer expectations of skills is shown in Figure 5. This figure shows that student expectations start in line with the framework at levels four and five, but seem to hit a ceiling there, remaining at that level for level six and seven courses. In contrast, lecturer expectations start significantly below the framework in level four courses and rise progressively, reaching alignment at level seven.

4.5 Overall Findings
The results presented in the previous sections suggest that lecturer’s expectations, although starting below the framework levels, rise more rapidly than students’ expectations. To give a clearer picture of this, Figure 6 shows the overall mean expectations of students and lecturers across all questions, compared to the framework levels.

As can be seen, lecturer expectations remain relatively stable at around level four across all cohorts. Conversely, lecturer expectations, although starting somewhat lower at 2.00, increase steadily across the levels to 6.04 at level seven, thus getting closer to, but not reaching, the framework level expectation.

5 Discussion
In this section, we discuss the implications of our findings for teaching. First, as shown in Table 2, we note that very few students chose the expected category. Moreover, in
almost all cases, the modal category did not represent the majority of students (Table 3). This poses a challenge for lecturers because there is no simple solution to the problem of what level to target for student activities.

Second, we note that student expectations remain broadly stable at around level four across all cohorts. This suggests that students do not expect the nature of skills, knowledge, self-management and group work to change as they progress through the levels. To improve alignment, educators should consider carefully the nature of coursework and learning activities to ensure that these are closely aligned with the framework. It is especially important to consider feedback mechanisms such as assessment in this regard.

Third, we note (see Figure 6) that, apart from starting somewhat lower at level four, lecturer expectations progress steadily through the levels. Broadly, they are between framework and student expectations at the higher levels. This pattern can be readily understood in the context of feedback mechanisms to lecturers. In these days of the consumerisation of education (Naidoo & Jamieson, 2005), most institutions use student surveys to evaluate the quality of teaching. This feedback mechanism will tend to bias lecturer expectations away from the framework towards those of the student cohort. Such feedback mechanisms are unlikely to change in the near future, so educators should consider fostering appropriate student-educator conversations to improve alignment.

Fourth, we note that the pattern of students’ expectations for self-management is low at level four and falls progressively at the higher levels. At level seven, it was significantly below the framework objective at level four: “self-management of learning and performance under broad guidance”. In this context, we note that the Ministry of Education states:

Given the significant investment the Government makes in students both through tuition subsidies and student support, students are expected to take responsibility for their own performance (Ministry of Education, 2013, p. 3.2).

Changing student expectations of self-management is likely to require the whole teaching team to take a consistent approach and actively promote expectations of self-management, and the associated benefits to students. Some practical measures could be wider use of self and peer assessment and involving students in setting appropriate framework-related learning activities and assessment criteria.

Fifth, as with self-management, students’ expectations of collaboration show a low and falling pattern across the levels with expectations of the level seven cohort below level four. It is interesting to note the objective at level four: “demonstrate some responsibility for the performance of others”. From our own teaching experience, it seems likely that students reject the validity of this, even though employers place a high value on working effectively in a team. This suggests that one way of modifying these expectations would be to expose students more to the values articulated by employers.

Sixth, although there is some evidence of an increasing trend, students’ expectations of the nature of knowledge are low and remain low. The students in the cohort at level seven have a level five expectation of knowledge at. At level five, the expectation is: “broad operational or technical and theoretical knowledge within a specific field of work or study”. In contrast, the expectation at level seven is “specialised technical or theoretical knowledge with depth in one or more fields of work or study”. From our own experience, we believe that students are too ready to carry out an internet search and copy and paste findings, considering this acceptable as knowledge. Activities that may be useful to effect change include requiring paraphrasing and summarising of material found, essay-type activities with compare and contrast, and embedding taxonomies such as SOLO (Biggs & Collis, 1982) into assessment rubrics. However, changing students’ expectations of the nature of knowledge will require a substantial “whole of team” approach.

Seventh, in contrast to the foregoing areas, students’ expectations of skills demonstrated appropriate expectations at level four and five, but seemed to hit a ceiling there. The framework expectation at level seven is to: “Analyse, generate solutions to unfamiliar and sometimes complex problems”. In contrast, the mean expectation of the level seven students was at level five: “Select and apply a range of solutions to familiar and sometimes unfamiliar problems”. Remediating this mismatch would require greater use in course work of unfamiliar and complex problems, and some unpredictable problems.

Overall, we believe that a concerted “whole of team” approach should be taken to align student expectations, and thus indirectly lecturer expectations, with those articulated in the framework. One way of achieving this would be to include a perspective of framework levels into regular course reviews. Learning activities, and especially assessed activities, should be mapped to framework levels to ensure alignment. It will be important to carry this out progressively, starting from lower levels, so that a student is presented with a coherent evolution of expectations as they progress with their study through the levels.

6 Conclusion

This study aimed to investigate the alignment of the expectations of students, lecturers and the NZQA. We carried out a systematic survey using an anonymous questionnaire to determine these expectations. We have presented our findings above. We summarise these findings in section 6.1, discuss threats to validity in section 6.2 and discuss our plans for further work in section 6.3.

6.1 Main Findings

Overall, we found that students’ expectations were significantly below the expectations set out in the qualifications framework. Moreover, they did not increase significantly as students progressed with their study to higher levels. Lecturer expectations were also below framework expectations, but exhibited an increasing pattern across the levels. Since lecturer expectations are necessarily a compromise between framework and student expectations, we believe that addressing the misalignment between student and framework expectations will indirectly result in a better alignment of lecturer expectations to the framework.
6.2 Threats to Validity
The lecturer sample size is too small to be considered fully representative of lecturer views. Nevertheless, it is sufficient to indicate that, for this sample, there is a systematic difference between lecturer expectations and that of the framework. The student sample size is adequate for our purposes. However, these data only represent a snapshot of student expectations at one point in time at a single institution.

Although the wording in the NZQA level descriptors was explicitly designed for communication to students, it is possible that some of our findings could be attributed to differences in interpretation.

6.3 Further Work
We plan to extend this study, in collaboration with researchers at other institutions, to determine whether our findings apply more generally to other institutions in the sector. We will also solicit additional lecturer expectations to help get a more representative sample of the lecturer perspective. We will also aim to extend this to a wider range of NZQA levels.

At this stage, our findings suggest that there is a misalignment between the various expectations at our institution, based on the framework descriptor wording. We will carry out further work to drill deeper into the reasons for this and then to carry out an action research programme to bring student and lecturer expectations into closer alignment with the framework. We will then repeat this survey to determine if we have achieved this goal.

7 References


Kemmis, R. et al., 2010. Transition from vocational education and training to higher education: Orientation, Wagga Wagga, NSW, Australia: Charles Sturt University.


Longitudinal Think Aloud Study of a Novice Programmer

Donna Teague
Queensland University of Technology,
Brisbane, QLD, Australia
Tel: +61 7 3138 2000
d.teague@qut.edu.au

Raymond Lister
University of Technology, Sydney,
Sydney, NSW, Australia
Tel: +61 2 9514 1850
Raymond.Lister@uts.edu.au

Abstract
Recent research from within a neo-Piagetian perspective proposes that novice programmers pass through the sensorimotor and preoperational stages before being able to reason at the concrete operational stage. However, academics traditionally teach and assess introductory programming as if students commence at the concrete operational stage. In this paper, we present results from a series of think aloud sessions with a single student, known by the pseudonym “Donald”. We conducted the sessions mainly over one semester, with an additional session three semesters later. Donald first manifested predominately sensorimotor reasoning, followed by preoperational reasoning, and finally concrete operational reasoning. This longitudinal think aloud study of Donald is the first direct observational evidence of a novice programmer progressing through the neo-Piagetian stages.

Keywords: Neo-Piagetian theory, programming, think aloud.

1 Introduction
Using neo-Piagetian theory, Lister (2011) conjectured there were four main stages of cognitive development in the novice programmer, which are (from least mature to most mature):

Sensorimotor: The novice programmer cannot reliably manually execute a piece of code and determine the final values in the variables (i.e., “trace” code). This incompetence is due both to misconceptions about programming language semantics and the inability to organise a written trace. Without the ability to trace accurately, and thus having no real capacity to check their own code, these novices can write incoherent code.

Preoperational: The novice can trace code reliably, but struggles to “see the forest for the trees”. That is, the novice struggles to understand how several lines of code work together to perform some computational process. When trying to understand a piece of code, such novices tend to use an inductive approach. That is, they may perform one or more traces with differing initial values, and make an educated guess based on the input/output behaviour. These novices also struggle to see the relationship between diagrams and code. When writing code, these novices tend to patch and repatch their code, on the basis of their results from tracing specific values through their code. They cannot truly design a solution.

Concrete operational: The novice programmer is capable of deductive reasoning. That is, the novice can understand short pieces of code by simply reading the code, rather than tracing with specific values. When reading code, they can abstract from the code itself to reason in terms of a set of possible values that each variable may have. These novices can design code, at least for algorithms that can be easily visualized as diagrams. However, novices at this stage tend to only reason about relatively short pieces of code that perform relatively familiar computational processes.

Formal Operational: Writing programs is frequently referred to as an exercise in problem solving. McCracken et al. (2001) defined problem solving as a five step process: (1) abstract the problem from its description, (2) generate sub-problems, (3) transform sub-problems into sub-solutions, (4) recompose, and (5) evaluate and iterate. It is only at the formal operational stage that novices can reliably and efficiently perform problem solving.

Levin (1986, p. viii) summarised the general change in the novice through these four stages (in any domain, not just programming) as being a process of:
1. Increasing logical-mathematical power;
2. Differing modes of representations – from perceptual to formal;
3. Increasing attentional scope and integrational ability; and
4. Increasing skill with applying the competencies of lower stages, along with the adoption of new strategies.

Corney et al. (2012) provided indirect evidence that novice programmers pass through the preoperational and concrete operational stages, by analysing student answers to questions in an end-of-semester exam. They found that (a) within individual exam questions, there were students who could provide a preoperational answer but not a concrete operational answer, and (b) across exam questions, students tended to consistently provide either a preoperational answer or a concrete operational answer. However, such indirect evidence does not indicate the actual thought processes of a student.

In this paper, we provide direct evidence that a student passes through these neo-Piagetian stages. We had several volunteer students complete programming related tasks while “thinking aloud” (Ericsson and Simon 1993). We met approximately once each week with these volunteers, so we could follow their progress over the course of a
Is concerned with the general cognitive development of children.

A child at a particular Piagetian stage applies the same type of reasoning to all cognitive tasks (e.g., math and chess), apart from exceptions known as décalage.

General tests, such as the pendulum test (Inhelder and Piaget 1958; Bond, 2005), can determine the Piagetian stage of an individual.

Prescribes typical age ranges for each Piagetian stage, but empirical evidence shows great flexibility in age ranges, due to cultural and environmental factors (Cole 1996, pp. 86-92).

Children spend an extended period in one stage, before undergoing a rapid change to the next stage – the “stair case metaphor”.

Is concerned with the cognitive development of people of any age as they learn any new cognitive task.

Since a person’s cognitive ability in any domain is a function of their domain knowledge, a person will often exhibit different Piagetian stages in different knowledge domains. Hence …

… there are no general tests, thus the failure to find strong correlations between programming ability and the pendulum test (e.g. Bennedsen and Caspersen 2008).

The time that individuals spend in any stage is free to vary, and varies according to their rate of knowledge acquisition in a specific knowledge domain.

The staircase metaphor is sometimes applied, but also so is the “overlapping wave” metaphor (Siegler 1996) – see Figure 1 and section 1.2.

Table 1: Classical versus Neo-Piagetian Theory

1.1 Classical versus Neo-Piagetian Theory

It is well known that researchers since Piaget have conducted experiments that call into question aspects of “classical” Piagetian theory. Less well known, however, is that modifications to Piaget’s classical theory have been proposed that address those experimental findings. One set of modifications is known as neo-Piagetian theory. (The “neo” is increasingly inaccurate, given that this “new” Piagetian theory is already several decades old.) Table 1 summarises some of the differences between classical and neo-Piagetian theory. For longer treatments of classical and neo-Piagetian theory, the reader is referred elsewhere (Demetriou, Shayer and Efklides 1992; Feldman 2004; Flavell, Miller, and Miller 2001; Lourenco and Machado 1996; and Sutherland 1992). In the next subsection, we will elaborate on the final row of Table 1, given that the concept of stages as overlapping waves is central to the empirical findings of this paper.

1.2 Stages as Overlapping Waves

Perhaps no aspect of classical Piagetian theory has generated more debate than the concept of stages. In classical Piagetian theory, children spend an extended period in one stage, before undergoing a rapid change to the next stage. Having made that change, children do not regress to the earlier stage. This is commonly referred to as the “stair case metaphor”. The stair case metaphor suffers from two broad types of problems. The first problem type is empirical – people have been observed to exhibit simultaneously the reasoning patterns of more than one stage. The second problem type is philosophical – how and why does a person make the quantum leap from one stage to the next? While some neo-Piagetian researchers still accept the stair case model, others have found evidence for the “overlapping wave” metaphor (Siegler, 1996; Feldman, 2004; Boom, 2004). That metaphor is illustrated in Figure 1.

According to the overlapping wave metaphor, as a person acquires knowledge in a new cognitive domain, the person exhibits a changing mix of reasoning strategies from different stages. Initially, the sensorimotor stage of reasoning is dominant, but its frequency of use declines. As the sensorimotor stage declines, there is an increase in the use of preoperational styles of reasoning, which

<table>
<thead>
<tr>
<th>Classical Piagetian Theory</th>
<th>vs.</th>
<th>Neo-Piagetian Theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is concerned with the general cognitive development of children.</td>
<td>vs.</td>
<td>Is concerned with the cognitive development of people of any age as they learn any new cognitive task.</td>
</tr>
<tr>
<td>A child at a particular Piagetian stage applies the same type of reasoning to all cognitive tasks (e.g., math and chess), apart from exceptions known as décalage.</td>
<td>vs.</td>
<td>Since a person’s cognitive ability in any domain is a function of their domain knowledge, a person will often exhibit different Piagetian stages in different knowledge domains. Hence …</td>
</tr>
<tr>
<td>General tests, such as the pendulum test (Inhelder and Piaget 1958; Bond, 2005), can determine the Piagetian stage of an individual.</td>
<td>vs.</td>
<td>… there are no general tests, thus the failure to find strong correlations between programming ability and the pendulum test (e.g. Bennedsen and Caspersen 2008).</td>
</tr>
<tr>
<td>Prescribes typical age ranges for each Piagetian stage, but empirical evidence shows great flexibility in age ranges, due to cultural and environmental factors (Cole 1996, pp. 86-92).</td>
<td>vs.</td>
<td>The time that individuals spend in any stage is free to vary, and varies according to their rate of knowledge acquisition in a specific knowledge domain.</td>
</tr>
<tr>
<td>Children spend an extended period in one stage, before undergoing a rapid change to the next stage – the “stair case metaphor”.</td>
<td>vs.</td>
<td>The staircase metaphor is sometimes applied, but also so is the “overlapping wave” metaphor (Siegler 1996) – see Figure 1 and section 1.2.</td>
</tr>
</tbody>
</table>
becomes dominant, before it in turn gives way to concrete operational reasoning. Not shown in Figure 1 is formal operational reasoning, which would develop in the same way. As will be apparent when we present the think aloud data for Donald, his progression fits the overlapping wave model.

1.3 Piaget vs. Bloom and SOLO

There have been earlier taxonomic descriptions of programming tasks, especially programming exam questions, based upon the popular Bloom’s Taxonomy (Bloom et al. 1956; Whalley et al. 2006) and the SOLO taxonomy (Biggs and Collis 1982; Lister et al. 2010). In this section we briefly justify our use of neo-Piagetian theory in preference to Bloom and SOLO.

Bloom is a taxonomy of questions, not a taxonomy of possible answers. That is, a question must be classified as belonging to a single level of Bloom’s taxonomy, and if a question is classified as being in one of the lower four levels of Bloom’s taxonomy, there is only a single binary decision to be made about a novice’s answer to that question — whether the answer is satisfactory or unsatisfactory at the prescribed Bloom level. Bloom is not suited to analysing questions where a population of novices may provide a rich variety of qualitatively different answers to a question. Nor does Bloom provide any mechanism for analysing think aloud data generated from the process by which a novice arrived at an answer.

The SOLO taxonomy is intended for classifying a rich variety of qualitatively different written responses to a question. However, SOLO does not provide any mechanism for analysing think aloud data generated from the process by which a novice arrived at a response. Biggs and Collis made a conscious design decision that SOLO was only for application to analysing final responses, not the mental process leading to that response (see pp. 21–23). Their reason for that decision was that they derived SOLO from classical Piagetian theory, and the restriction of SOLO to analysing final responses was their approach to avoiding the problems with classical Piagetian theory that were discussed earlier in this paper. Note that Biggs and Collis published SOLO in 1982, before almost all the developments in neo-Piagetian theory that provide an alternative way of avoiding the problems with classical Piagetian theory. Unlike SOLO, neo-Piagetian theory preserves the Piagetian mechanisms for analysing think aloud data generated from the process by which a novice arrived at a response.

Furthermore, given SOLO’s focus on responses to questions, and the conscious exclusion from SOLO of the process by which a response is generated, SOLO does not lend itself to generating ideas for questions to put to students in think aloud sessions, whereas neo-Piagetian theory (through concepts such as reversibility, conservation and transitive inference) has proven to be a rich source of inspiration for us. All the problems we put to Donald (apart from tracing problems) were inspired by problems that Piaget used on children. The observations that Piaget made using his problems also provided strong suggestions as to what to look for in Donald’s think aloud sessions.

We regard our use of neo-Piagetian theory as a logical progression from the earlier research that used SOLO.

1.4 N = 1?

Some readers may be disturbed by our small sample size – a single student. To argue for a larger sample, however, is to argue from a positivistic perspective, which is not a wrong perspective, but it is a perspective orthogonal to the aims of this paper. Our research is qualitative, not quantitative. That is, our aim is to identify some aspects of the nature of how novices reason about programs. Our aim is to neither identify all aspects of how novices reason about programs, nor to count the frequency with which a particular aspect occurs in a population of novice programmers.

Our use of think aloud sessions is an example of the microgenetic research method, which has been applied in many domains to test theories of cognitive development, and which is defined as having three main properties (Siegler 2006, p 469):

1. Observations span the period of rapidly changing competence.
2. The density of observations is high, relative to the rate of change. In the first semester of this study, think aloud sessions were conducted once a week (although for space reasons we only report three such sessions in this paper).
3. Observations are analysed intensively, to infer the representations and processes used by the students.

The microgenetic method has been used previously by Lewis (2012) to study a single novice programmer. We regard our research, and the earlier research of Lewis, to be a necessary prelude to conducting quantitative research. That is, we regard our work as the identification of interesting aspects of a novice programmer, which may then be studied quantitatively, either by us or by other researchers.

2 Week 3: Tracing Code

Each think aloud session with Donald was recorded with a Livescribe Smartpen (2013) which captured everything that Donald wrote and spoke. The scripts that Donald completed were then processed to produce “pencast” PDFs, the audio-synced video contents of which are repla-playable using Adobe Acrobat Reader. The audio was also transcribed. Ellipses (“...”) are used throughout the transcripts to indicate both missing utterances which add little to the context (for example, sighs, laughs, coughs, and fillers such as “um”, “mummy”, and “huh”), and also short pauses in articulation.

The first think aloud task performed by Donald is shown in Figure 2. Donald performed this task in week 3 of semester, but he had already discussed this problem with his lecturer, at a one-on-one meeting. The lecturer had shown Donald a way to perform a systematic trace on that code.

Donald began by writing out the code as shown in Figure 3. The left hand sides of lines 1 to 5 and also lines 6 to 10 are the code from Figure 2. The right hand sides of lines 1 to 5 were subsequently written by Donald as he updated variables during his trace. As we shall see, writing those updated values on the right hand side may be one source of his subsequent confusion during the trace.
As he began his trace, Donald recalled that his lecturer had used a systematic way to record a trace:

I remember there was an easier way to do this, visually... a way to write this out to make it very easy to represent.

He then started tracing the code from line 6, writing the new values stored in each variable next to the first block of given code (i.e., lines 1 to 5). This was NOT the layout that the lecturer had demonstrated to Donald. Note that, in starting at line 6, Donald ignored line 5. As he wrote "a = 3" on the right hand side of line 1, Donald said:

So immediately, well if a equals b, a equals 3.

He then looked at line 7 and said:

b equals e. ... ah ... that change it? No. ... b = 3, so b right now equals a, which equals now 3.

From the transcript of the think aloud session, it is not clear what Donald meant by “that change it?” However, a year after Donald performed this think aloud, we had him listen to this pencast, and he explained that at that stage of his development he had been unsure of what assignment statements did:

My confusion with [for example] e = a was due to me not realising that e received a copy of the value of a and then they are separate. I thought they were still connected.

That novice misconception about assignment statements – that it “connects” variables – is well known (e.g., Du Boulay 1989). As a consequence of that misconception, Donald incorrectly wrote on the right of line 2, “b = 3”.

That is, he reasoned (incorrectly) that e had been “connected” to a at line 5, then a had been “connected” to b at line 6, so at line 7 b was effectively set to itself – hence his “no” in response to his own question “that change it?”.

Donald then correctly evaluated lines 8 and 9, writing that “e = 2” and “c = 4”. At line 10, he began correctly:

... c equals d ... which is 4, and d equals e ... which equals 2.

But then Donald added:

... if I go to the updated version

Donald was not sure he should be using the “updated” value “e = 2” as he wrote on the right hand side at line 3, or whether he should follow a similar chain of reasoning as before – that line 8 “connected” variables e and c and line 9 connected c to d. To follow that chain of reasoning would be consistent with how Donald (incorrectly) evaluated line 7, but in writing “d = 2” on the right hand side he elects to not be consistent. Donald then revisited his trace, to check for errors:

So just to revise ... a starts off equalling 7. However it needs to equal b which equals 3. ... b equals 3. Hang on, but b equals e ... which equals a, which equals 3. Yeah ... ok ... e ... Oh! I totally missed that ...

By “I totally missed that”, Donald was indicating that he had not previously noticed line 5, “e = a”: e equals ... a, which equalled 3. ... [sound of air being sucked through teeth] ... Yeah, because e has already been changed.

Having become confused, Donald restarted his trace from the beginning. The following extract from the transcript may not be coherent, but it does illustrate his growing confusion:

a equals b, and b equalled 3 ... b equals e, and e equalled a, which then became 3 ... so then e equals c ... e equals ... a...arrgh...so then c becomes 3. ... No it doesn’t. Does it? Arrgh!

But Donald showed determination. He started another review of his trace, and when he became confused, he restarted yet again. He began that trace by reciting the first four lines of code:

... a has been assigned 7, b has been assigned 3, c has been assigned 2, d has been assigned 4.
For these four lines, his choice of the word “assigned” was a change from his earlier use of “equals”. He then articulated line 5 differently, using the word “equal”:

... and e has been said to be equal to a, which is 7. So it’s currently 7 ....

Thus in reciting lines 1 to 5 he articulated (a) the correct conception that a constant value to the right of an equals sign indicates an assignment of that constant to the variable on the left hand side, but (b) the misconception that variables on both sides of an equals sign “connects” the two variables.

Donald then continued on to correctly evaluate line 6, but at line 7 his misconception about line 5 lead him astray:

... b now has the value of e, e has the value of a, and I changed the value of a, so that makes it have the value of also 3.

At this point, Donald had been working on this problem for 7 minutes and 18 seconds. He continued for another two minutes, while becoming even more confused, before writing his final answer. Donald expressed low confidence in his final answer:

I still think that is wrong. I’m not really sure. I think it’s sort of right.

In fact, Donald’s answer happened to be correct for all the variables except b, for which he had the value 3 instead of the correct value of 7. However, had Donald consistently applied his misconception that variables on both sides of an equals sign “connects” the two variables — then lines 5 to 10 should have “connected” all the variables, in which case all the variables would then have the same value. Near the end of his trace, Donald actually made that same point:

... I thinking I’m just changing everything to 3 now by accident but we’ll see what happens.

While Donald’s misconception about assignment statements has been well known for decades, what we see in Donald’s think aloud is that he does not apply that misconception consistently. According to neo-Piagetian theory, such inconsistency is common in novices reasoning at the sensorimotor stage.

Later in this think aloud session, Donald reflected on his ad hoc approach to recording his trace, especially his recording of variable values on the right hand side of lines 1 to 5:

... I represented it the wrong way. I probably should have had this be more ... like ... move it down so it is in a line ... rather than try to do this and then go back up.

When we interviewed Donald a year after he had done this think aloud session, he reflected on the problems he was having at the time he did this trace:

... it takes me a very long time to remember how to think like a computer, and that’s really what I find slows me down, because my mind wants to try and handle it a different way – but I’m like “No, a computer! You go line by line” ... but to me that’s not the first way my mind wants to work ... I don’t have that automatic ... a computer is very simple actually. Looks like it is very complicated ....

In summary, at this week 3 think aloud, Donald displayed the characteristics of a programming novice working at the sensorimotor stage. The misconceptions he had about programming concepts were applied inconsistently. He was cognitively overloaded on a simple tracing task as he was unable to organise an effective and accurate method for tracing code.

In classical Piagetian theory, the sensorimotor stage is experienced by infants. In the application of neo-Piagetian theory to novice programmers, the use of the term “sensorimotor” to describe the initial stage remains appropriate, since at this stage the novice programmer has trouble interpreting the semantics of the code he or she is reading (i.e. the sensory component) and also has trouble with writing down a well organised trace (i.e. the motor part). Furthermore, the sensory and motor components interact. For example a misconception about what a piece of code does can lead to an incorrect method of recording within a trace the result of applying that misconception.

3 Week 9: Explaining By Tracing

At his week 9 think aloud session, which was his fifth such session, Donald attempted the explanation problem in Figure 4. We have already described this particular think aloud by Donald in an earlier publication (Teague et al. 2013). Here we summarise those aspects of the think aloud that are most salient to this paper.

Donald attempted to explain the code by using the inductive approach of a novice at the preoperational stage of neo-Piagetian theory. That is, he selected some initial values for the variables \((y_1 = 1, y_2 = 2 \text{ and } y_3 = 3)\), then traced the code with those values, and then inferred what the code did from the input/output behaviour. However, at week 9, Donald was still having some problems organising his trace, so his use of the inductive approach did not initially go smoothly.

Donald began with the unsuccessful trace shown in Figure 5. Like his week 3 trace, this trace was not well organised. Each of the three lines of that trace represents an if-then block from the code in Figure 4. The numbers and arrows are Donald’s attempt to record how the values in the variables change as the code is executed. Unlike his week 3 trace, the transcript of this week 9 trace shows that Donald had a correct and consistent understanding of how the code works. But by the time he reached the third line of that trace, his method of recording the values led him to confusion.

Donald then attempted a second, more organised trace, as shown in Figure 6. He first wrote, on each of the three lines, respectively \(y_1 = 1\), \(y_2 = 2\) and \(y_3 = 3\) (the numbers “1”, “2” and “3” were subsequently crossed out as his trace progressed). He then performed a conventional and correct trace, which took him only 67 seconds. In performing this second trace, Donald showed clear progress from the haphazard sensorimotor approach he used in week 3.

However, based on this one successful trace, Donald then made an incorrect inductive inference, which led him to write the following incorrect answer:

“To reverse the values stored in \(y_1, y_2\) and \(y_3\) ...”

We then asked Donald to trace the code again, using the initial values \(y_1 = 2, y_2 = 1\) and \(y_3 = 3\). He performed a
successful trace with those values, using the same approach as in his previous trace. On completing this trace with our values, however, Donald initially maintained that this trace confirmed his initial answer, with this trace having “ended up the same ... as what I originally came up with”. (Although his tone of voice in the recording might indicate uncertainty, or irony.) After being challenged by us, but without us providing any further hints, Donald exclaimed:

“Oh! I’m ordering them ... um ... so, it’s more about, it’s not to rev ... hang on ... oh [indecipherable]... rather than to reverse, it would be to, place them from highest to lowest.”

In this week 9 think aloud, Donald initially showed attributes of the sensorimotor stage, but he then went on to also show some of the attributes of the preoperational stage. After an initial unsuccessful trace, he performed two well organised and successful traces. However, using the inductive approach based on the input/output behaviour of his first successful trace, Donald jumped to a rash and incorrect answer. This answer was especially rash, because the initial values he chose resulted in all the if conditions being true. (Some of the other students who participated in our think aloud study did carry out an initial trace with the same values chosen by Donald, but they also carried out a second trace with different values.)

However, when Donald was prompted to perform a second trace, with values given to him by us, he did infer a correct description of the purpose of the code. Donald manifested behaviour consistent with someone who, in terms of the overlapping wave metaphor, is transitioning from the sensorimotor stage being dominant to the preoperational stage being dominant. In this week 9 think aloud, Donald did not manifest any aspect of concrete operational reasoning.

3.1 The Concrete Operational Approach

Instead of reasoning about the Figure 4 problem in terms of specific values, as Donald did, a novice reasoning at the concrete operational stage would tend to reason (albeit implicitly) about the code in terms of algebraic constraints on the values in the variables. For example, after the first if statement in the code from Figure 4, the concrete operational novice would think of \( y_1 \) as holding any possible value that satisfies the condition that it is less than the value in \( y_2 \). After the second if, the concrete operational novice thinks of \( y_3 \) as holding any possible value that satisfies the condition that it is less than the values in both \( y_1 \) and \( y_2 \). By thinking in this deductive fashion, the concrete operational student feels little need to understand code via the inductive, preoperational approach of tracing specific values.

4 Week 13: Abstract Reasoning

In neo-Piagetian theory, one of the defining characteristics of the concrete operational novice is the ability to reason about abstract quantities that are conserved. For example, in a classic Piagetian experiment, a preoperational child believes that when water is poured from one container into another, and the
water level is higher in the second container, then there is more water in the second container than there was in the first. In contrast, an older child at the concrete operational stage is aware that the quantity of water is conserved.

In a programming context, Lister (2011) conjectured that a preoperational programming student would tend to think that small changes to the implementation of an algorithm would change the specification of what the code does. Equally, Lister argued that a concrete operational student should be able to make small implementation changes to code while conserving the specification. He nominated a problem like that shown in Figure 7 as an example of a problem that requires a concrete operational understanding of programming. We had Donald attempt this problem at his week 13 think aloud session.

Figure 8 shows Donald’s attempt at this week 13 task. (Note that Donald wrote his answers for this task on a blank page. We have superimposed his answers over the question text in Figure 8, and retained the sample answers in boxes on the right of Figure 8, to make it easier for the reader to follow.)

In the three boxes in Figure 8, Donald only provided correct code for one box. However, neo-Piagetian theory merely says that a student progresses from sensorimotor, to preoperational to concrete operational when the programming constructs to which the novice is exposed do not change. When new programming constructs are introduced (as loops and arrays are in the week 13 problem) then a novice may need to pass through the neo-Piagetian stages for these new constructs. Thus, a student may have a concrete operational grasp of non-iterative and non-array aspects of a piece of code, but at the same time be reasoning about the iterative/array aspects at the preoperational or sensorimotor stage. On inspection of the transcript for Donald’s attempt at the week 13 problem, it is obvious that he struggled with the distinction between a position in an array and the contents of that position – as many novices do when they first encounter arrays. Therefore, with respect to arrays, especially when writing code, Donald is at the sensorimotor stage of development.

However, a close inspection of the transcript shows that Donald has made some progress since week 9 with reasoning about other code. The remainder of this section will emphasise the transcript evidence for the progression in aspects of Donald’s reasoning.

As Donald began by reading the problem, he articulated a clear understanding of what was required, and a clear acceptance that two different implementations might satisfy the same specification:

_So it does the same thing, but is going to be doing slightly different code because ... line 5 is different._

As Donald read through listing 1, he articulated an imprecise description of line 2, indicative of his weakness in distinguishing between a position in an array and its contents:

... _x zero is best ..._

In the **Source Code Listing 1** below is code for a function which returns the smallest value in the array _x_. When the code in **Source Code Listing 2** below is correctly completed, it should also return the smallest value in the array _x_. Line 5 is different in the two listings. Except for line 5, and the lines with boxes, all other lines in the two listings are the same.

**Source Code Listing 1**

```java
1. public int Min(int[] x) {
2.     int best = x[0];
3.     for (int i=1; i<x.Length; i++){
4.         if (x[i]<best) {
5.             best = x[i]; // different from line 5
6.         } // in the second listing
7.     }
8.     return best;
```

**Source Code Listing 2**

```java
1. public int Min(int[] x) {
2.     int best = 0;
3.     for (int i=1; i<x.Length; i++){
4.         if (x[i] < x[best]) {
5.             best = i; // different from line 5
6.         } // in the first listing
7.     }
8.     return x[best];
```

After reading lines 3, 4 and 5 of listing 1, Donald then summarises the entire loop in a way that shows some nascent signs of concrete operational reasoning:
Ok, so it's just going through the list ... so every time it finds something smaller it assigns to best until we get to the end ...

Had Donald then added something like “so at the end of the loop best will contain the smallest value in the array” that would have been unambiguous evidence for concrete operational reasoning, but what he actually uttered is at least a coherent summary of the four lines of code that form the loop.

Donald then read listing 2. He briefly adopted a quasi-preoperational approach to reasoning about that code, by considering how the code in the second listing would work for the specific case of the first iteration of the loop:

If \( x \) at position 1 is less than \( x \) at position 0, it would take the element number of \( i \) ... and then assign it to best. Then element one … has the least.

Note, however, that while Donald considered two specific positions in the array, he did not consider specific values at those positions. Nor does he consider any other specific positions in the array. He appears satisfied that his solution was correct.

For the array values, he chose 2, 1, and 3, in that order. In placing those array values in that order, Donald demonstrated his sensorimotor difficulty in distinguishing between the position in an array and the contents of that position. After we had helped him correct his answer for the first box, he immediately corrected the third box without any help from us. In so doing, he showed some nascent concrete operational understanding of the relationship between the code required in the first and third boxes.

In summary, Donald’s weakness with arrays was obvious in this week 13 task. However, if his weakness with arrays is ignored, then there are signs in the week 13 task that he had begun to progress beyond the exclusively inductive approach he used in week 9. That is, he showed some capacity to reason about code without needing to perform a complete trace with specific values.

5 Concrete Reasoning

Figure 9 shows Donald’s attempt at the problem in Figure 7 three semesters after his attempt shown in Figure 8. He was by then nearing completion of his degree and had successfully completed six programming courses. Donald approached the task with confidence:

… should be in principle pretty easy to do. So if I look at the first code public int min, so pass in the array ... then we just iterate through incrementally ... and if the current is less than best, we pass that in.

While Donald did, in the above transcript extract, articulate three keywords (i.e. public int min) as he often did in earlier think alouds, here he went on to articulate an abstraction beyond just the keywords, for example:

so pass in the array instead of “int x”;

then we just iterate through incrementally instead of articulating the lexical symbols on line 3;

and if the current is less than best instead of articulating the lexical symbols on line 4, with his use of “current” suggesting an abstraction beyond the code itself, which is consistent with a subsequent articulation of the for loop at line 3:

... int i ... is assigned 1 and then it keeps going through ... the length of the array

Donald has given a reasonable explanation for the functioning of the for loop. He has done this in abstract terms, rather than relying on specific values of elements or indexes to explain what’s going on. Earlier in his development, as described in the previous section, Donald’s behaviour had been more pre-operational, and he had relied on specific index positions when he talked about the same looping structure (i.e., if \( x \) at position 1 is less than \( x \) at position 0, it would take the element number of \( i \)). Donald now further demonstrates that he has developed an ability to explain code in an abstract manner:

so if the current element of \( x \) is less than best, \( i \) ... which is the value of that, is put into best.

In the above transcript extract, it is unclear whether Donald is thinking of the variable best as being a value copied from the array, or best as representing a position in the array. However, he begins to improve the clarity of
Unable to reason deductively about code or see a relationship between different parts. Then finally we saw evidence of his transition into the concrete operational stage where he can reason and explain the purpose of code, talk in terms of abstractions rather than specifics and consider consequences and alternatives.

At this stage of our research programme, the question remains as to whether Donald represents a significant portion of novice programmers. Based upon our work with other students, we suspect he is not a rare exception, but that will need to be confirmed by quantitative research.

The computing community has tried many variations on how to teach programming, but many students continue to struggle. Neo-Piagetian theory points to one aspect of programming pedagogy that has remained largely invariant across those many past variations – our teaching skips too quickly across the sensorimotor and preoperational stages for many students. We suggest that teaching be designed explicitly with students’ current level of reasoning in mind. As the cognitive skills developed through the neo-Piagetian stages are sequential and cumulative, novices need to be reasonably strong at a lower neo-Piagetian stage before they can be expected to reason well at a higher neo-Piagetian stage. Otherwise, teachers are in danger of talking to their students in a way that the students are not yet capable of processing.

Some computing academics claim that students who struggle to learn programming lack an innate talent for programming. Any readers of this paper who share that suspicion might think that Donald’s early performance in think alouds indicated that he lacked such a talent. Those readers may be surprised to learn that Donald has completed his degree with a high grade point average (more than 6 out of a possible 7), and is, in the near future, taking up a fulltime position at an international corporate professional services firm as a business IT consultant. Donald’s academic achievements may indicate that programming ability is something that is learned, rather than something innate. Donald remained enthusiastic and determined no matter how hard he found the tasks we gave him. He saw those tasks as learning experiences, and consequently he improved. Perhaps Donald personifies the primary qualities required to learn programming – perseverance, a desire to learn – not an innate ability to program. What instructors need to do is provide instruction targeted at an appropriate level of abstract reasoning for their student(s), rather than assume that students have the cognitive maturity to perform programming tasks requiring concrete operational reasoning.

7 Acknowledgments

Our heartfelt thanks and congratulations to Donald, whose approach to learning is an inspiration to us. Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office for Learning and Teaching or the Australian Government.
8 References


Program visualization and explanation for novice C programmers

Matthew Heinsen Egan¹  Chris McDonald²

School of Computer Science and Software Engineering
University of Western Australia
Crawley, Western Australia 6009
Email: ¹m.heinsen.egan@graduate.uwa.edu.au ²chris.mcdonald@uwa.edu.au

Abstract

Program visualization and natural language explanations of program behaviour have been shown to assist novice programmers with improving their programming knowledge, correcting misunderstandings, and debugging programs. These techniques have been used in several novice-focused debugging systems, but few have been developed for the C programming language—despite it being widely reported as a difficult language for novices. We present robust, maintainable systems for visualizing the memory state and explaining the behaviour of programs written in the standard C programming language.

Keywords: Novice programmers, debuggers, visualization

1 Introduction

The standard C programming language can be especially difficult for newcomers. In particular, pointers and manual memory management can present difficulties both in understanding at a conceptual level, and in debugging the laconically described runtime errors which result from their misuse. Most newly developed novice-focused debugging systems are designed for object-oriented programming languages, as introductory teaching has focused on these languages, and the most notable tools developed to assist novice C programmers are predominantly unmaintained.

Research from the fields of programming languages and compilers has developed many advanced debugging techniques, but they are typically only supported by tools designed for expert programmers, rather than for novices. The complexity of these tools, and the time required to learn their use, at even a modest level, are often insurmountable hurdles for novice students. Furthermore, while these tools can be used to locate runtime errors, they do not assist novice programmers to understand those errors, or more generally to understand the behaviour of their programs.

Several novice-focused tools have implemented graphical program visualizations and automatically generated explanations of program behaviour. These features have been shown to assist novice programmers with constructing knowledge and debugging programs in numerous evaluations, such as those described by Brusilovsky (1993), Smith & Webb (1995), Moreno & Joy (2007), and Cross et al. (2009). However, few of these tools have supported the C programming language, and those that do are typically incomplete or unmaintained.

In this paper we introduce novice-focused systems for creating graphical visualizations of the runtime memory state of C language programs, and for generating natural language explanations of C program fragments. Our systems are designed to be robust and reusable. They build upon a previously developed novice-focused debugging system for the C programming language, augmenting the existing runtime error detection and execution tracing with program visualizations and natural language explanations.

The remainder of this paper is organized as follows: Section 2 discusses prior work in this area, Section 3 describes the project that acts as the foundation of our work, Section 4 describes our graphical visualization system, Section 5 describes our system for generating natural language explanations, Section 6 discusses the integration of these systems into the foundation project. We finally summarize our discussion and highlight future plans in Section 7.

2 Prior work

Zimmermann & Zeller (2002) introduce a tool that automatically extracts memory graphs from a program. Their tool extracts information about a program’s memory state using the GNU Project Debugger—a free, open source debugger that supports many languages and platforms¹. The system is not designed for novice programmers, but they discuss some of the challenges involved in automatically creating graphs from the memory of C language programs. These challenges are rarely discussed in relation to novice-focused tools, though they still exist in novice programs. A summary of their discussion of the most prevalent issues follows:

Invalid pointers. In C a pointer may reference invalid memory. To dereference such a pointer would introduce garbage into the graph. Their system determines valid pointers by querying the debugger to find valid memory areas.

Dynamic arrays. Dynamic memory allocations can be used to allocate arrays of arbitrary size. C has no standard means to find out how many elements were allocated, thus any analysis of a program’s memory must determine this itself. Their solution is to query the debugger to find the size of the memory area that is occupied by the array, and determine the maximum number of elements that will fit within this area.

¹http://www.sourceforge.org/gdb/
Unions. C has no standard method for determining which member of a union is active. Zimmermann & Zeller attempt to select a single member to use when constructing a memory graph: “To disambiguate unions, we employ a couple of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a type tag – an enumeration type within the enclosing struct whose value corresponds to one of a union member. While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules either hand-crafted or inferred from the program.”

VIP is a novice-focused program visualization system that supports a subset of the C++ programming language, presented by Virtanen et al. (2005). It displays the evaluation of each statement in detail, and supports reversible visualizations. Example programs can be embedded with special inline comments, hidden from the user, which can provide explanations at certain points of execution. VIP uses a custom interpreter and is designed for use only with small programs. It was not formally evaluated, but it was made available to students in an introductory programming course whose assessment of the system was altogether positive according to a questionnaire performed near the end of the course.

Hundhausen & Brown (2007) described ALVIS, a “radically dynamic” programming environment: each change to the program causes the system to re-parse the code and dynamically update the accompanying program visualization. ALVIS supports only a subset of the C programming language. This reduces the difficulties of visualization, but also limits the usefulness of the system. Usability studies performed with novice programmers indicated that ALVIS is useful for debugging.

HDPV is a data structure visualization system for programs written in C, C++, or Java, presented by Sundararaman & Back (2008). In HDPV’s design, language-dependent program monitors send information to a language-independent visualizer, which displays the monitored program’s runtime state using a force-directed graph layout. Two monitors are described: a monitor for C/C++ programs, which uses binary instrumentation; and a monitor for Java programs, which uses bytecode instrumentation. The visualizer is implemented using the prefetch toolkit, and allows the user to manipulate the visualization by panning, zooming, repositioning nodes, or editing sections of the graph. The visualizations are intended to be usable for identifying errors in the program’s runtime state, such as buffer overflows or memory leaks, or for identifying logical errors in the program’s data structures. HDPV’s effectiveness has not been evaluated, and it appears to be unavailable. There is no discussion, or example, of handling the difficulties of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a type tag – an enumeration type within the enclosing struct whose value corresponds to one of a union member. While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules either hand-crafted or inferred from the program.”

HDPV’s effectiveness has not been evaluated, and it appears to be unavailable. There is no discussion, or example, of handling the difficulties of visualization in relation to those topics. The authors describe the development of “focused visualization environments” to explore specific topics in detail. Our work is more general in that it is designed to assist novices with debugging their own programs, however, it does intrinsically visualize many of the topics most frequently considered to be critical or difficult, such as parameter passing, recursion, scope, pointers, and memory allocation. Despite the rarity of combining graphical visualizations with natural language explanations, the survey found this to be a desirable feature: “The majority of respondents (89%) felt enhancing graphical visualization with textual visualization would help improve the value of visualization.”

Brusilovsky (1993) formally evaluated the debugging effectiveness of program visualization with ITEM/IP-II. This program visualization system supports an educational mini-language named Tortoise, and generates textual explanations of program execution. The evaluation’s subjects were 30 students, who used the ITEM/IP-II system to solve problems in their introductory programming course. When a student’s solution was in error, they were given an increasing amount of assistance until they understood the location and source of the bug; firstly, knowledge that there is an error; then the results of the student’s program and a model program, on the test that produced the error; then the visual execution of the student’s program on the test that produced the error; then a lab assistant vocally simulating explanatory visualization; finally the lab assistant would attempt to explain the error to the student. Students only required the lab assistant’s explanation in 16% of cases. Visualization and simulated explanatory visualization effectively assisted students in 39% and 20% of cases, respectively.

Explanatory program visualization also features in Bradman, a system designed to assist novice programmers learning C, presented by Smith & Webb (1995). Bradman is a visual interpreter which “assists the user by giving him/her a visible model of the workings of the program” and an “explicit, detailed explanation of the effect of each statement as it is executed.” Experimental evaluation of Bradman’s explanatory visualization, wherein students used Bradman either with or without the feature, showed that students with access to the feature felt more strongly and more often that Bradman assisted them in finding bugs.

The benefits of explanatory systems are intuitive: many bugs arise from an incomplete or incorrect understanding of the programming language, and a natural language explanation of the source code can enable students to gain or correct the knowledge that is necessary to understand and correct such bugs. Previous explanatory systems for the C programming language have relied on custom parsing solutions. Such systems are susceptible to incompletely supporting the language, due simply to the size and complexity of the task. Using custom parsing implementations also reduces the ability to reuse the explanatory system in other tools, and increases development and maintenance costs.

3 SeeC

Our work extends the SeeC project introduced by Heinsen Egan & McDonald (2013a): a novice-focused system for the standard C programming language that provides execution tracing and runtime error detection. SeeC itself is built upon the Clang project2: 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. This provides SeeC with robust support for the C programming language while avoiding the unsustainable maintenance costs.

2http://clang.llvm.org
FILE pointers allow clients to get the raw value of the pointer, and to determine whether or not the pointer is valid (i.e. whether or not it references a currently open FILE stream).

4 Graph Visualization

Our system for graph visualization is built upon SeeC’s representation of recreated states (described in Section 3). It operates on a single Process State and produces a graph in the DOT language. We will not describe the language in detail (for more information see the Graphviz website\(^1\)), but it is important to describe one feature that our system uses extensively: “HTML-like” labels.

An HTML-like label allows a graph node’s label to be described similarly to an HTML table element. We use this to render related values within a single graph node, e.g. in Figure 1 there are three nodes: one for the function main, one for the function getright, and one for a block of dynamically allocated memory. Edges can be attached to specific cells inside the labels. This allows us to produce concise graphs while accurately representing the source and destination of pointers.

Figure 1: Pointers to struct and member

We previously discussed a number of difficulties with generating graphs of C programs’ memory that were described by Zimmermann & Zeller (2002). Some of these issues are effectively handled by SeeC’s representation, in particular the validity of pointers and the size of dynamic arrays are already determined by the underlying system. We do not attempt to unambiguously display unions, rather we simultaneously display all members of the union so that students can examine their behaviour. However, pointers can also cause a region of memory to have conflicting interpretations, and in this case we do attempt to reduce ambiguity by showing a single interpretation of memory, the exact process of which we will describe later in this section.

The first stage of our graph generation system is to inspect all Values in the Process State. We recursively inspect all elements of arrays, all members of records, and all dereferences of pointers. During this process we record all pointer relationships into an object called the Expansion.

The next stage is to generate the layout for all global variables, threads, and memory areas. Each of these layouts can be generated independently of the others. A layout contains the label of the node in the

---

\(^1\)http://www.graphviz.org
DOT language, the node’s identifier, the memory area that the node represents. It also contains information for where edges should be attached for each Value that is represented in the node. This will be used in the final stage to create edges for all of the pointers in the state.

Each thread is represented by a sub-graph which contains the nodes for each function in the thread’s call stack. These nodes are aligned horizontally and ordered according to the order of the call stack. A Function State’s label has a title row containing the name of the function. This is followed by one row for each parameter and local variable, with the name of the parameter or local variable occupying a cell on the left, and the Value occupying a cell on the right. A Value’s label contents are generated by a Value Layout Engine. The graph generation system supports multiple Value Layout Engines, and allows students to specify which engine should be used for any particular Value. We can also provide new Value Layout Engines, provided they implement the appropriate interface. Engines are not required to handle all potential Values: the engines may be queried to determine whether or not they are capable of performing the layout for a particular Value. The default behaviour is to use the first engine that is capable of performing the layout for each Value. If the student has specified a particular engine to use for a Value and that engine reports that it cannot perform the layout, perhaps because some property of the Value has changed, then the graph generator will fall back to the default behaviour.

The default Value Layout Engine is capable of performing the layout for any Value. It generates the layout based on the particular kind of Value, as follows:

**Scalar** Fill the cell with the string description of the Value.

**Array** Create a new sub-table in the cell, with two columns, and one row for each element in the array. Place the index of the elements in the left column’s cells, and then recursively layout the right column’s cells using the elements’ Values.

**Record** Create a new sub-table in the cell, with two columns, and one row for each member of the record. Place the names of the members in the left column’s cells, and then recursively layout the right column’s cells using the members’ Values.

**Pointer** If the pointer is uninitialized then fill the cell with the placeholder “?”. If the pointer’s raw value is zero then fill the cell with the text “NULL”. If the pointer has no valid dereferences then fill the cell with the placeholder “+”. Otherwise, leave the cell empty – it will be connected appropriately when edges are created.

The process for generating the layout for a memory area begins with selecting which type should be displayed, because a memory area may have multiple references of differing types. This does not necessarily constitute an error. Selection of the type proceeds in the following manner:

1. Remove all void pointers from the list of references. If there are no other references, then layout the memory area as void.

2. Remove all pointers to incomplete types. If there are no other references, then layout the memory area using the incomplete type.

3. Remove all pointers which reference the child of another pointer’s dereference. This handles situations such as the program in Listing 1, where a memory area is referenced by both a pointer to a struct and a pointer to one of that struct’s members. A visualization of this program was shown in Figure 1.

4. If the remaining pointers have the same type, then perform the layout using this type. Otherwise we layout using one of the conflicting types (the other references will appear type-punned). Alternatively, we could render all types simultaneously and use a visual cue to indicate that they occupy the same memory, or we could elide all of the types and instead display an information message indicating that multiple conflicting types are referenced in the area.

**Listing 1 Pointers to struct and member**

```c
#include <stdlib.h>

typedef struct {
    int left;
    int right;
} FOO;

int getright(FOO *fooptr) {
    int *iptr = &fooptr -> right;
    return *iptr;
}

int main() {
    FOO *ptr = malloc(sizeof(FOO));
    *ptr = (FOO){ .left = 1, .right = 2 };;
    getright(ptr);
    return 0;
}
```

After a reference has been selected, area layout is performed by an Area Layout Engine. The operation is analogous to a Value Layout Engine, allowing us to create special rendering for certain types of Values. For example, we use a layout engine for C strings to condense the display into a horizontal representation. We can see this in Figure 2: the default representation of an area with multiple dereferences is to display indices on the left and values on the right, as shown by the argument vector, whereas the C string representation is used for the arguments, allowing a more natural representation. The C string representation also allows us to elide values that follow the terminating null byte.

![Figure 2: C string layout](image-url)
contains the memory occupied by the pointer, and the layout of the node that contains the address referenced by the pointer. For example, consider the variable \( \text{ptr} \) displayed in Figure 1: the memory occupied by the pointer is contained by the node of \texttt{main}, and the referenced address is contained by the node of the dynamically allocated memory. Next we will search the layouts to determine where the tail and head of the edge should be connected. If we cannot find a connection for either the tail or head of a pointer then we connect the edge to the node, and adjust the end of the edge to indicate that the value is not rendered in the graph (currently this is represented by using a circle rather than an arrowhead).

5 Explanations

Previous studies have shown that automatically generated natural language explanations of program source code can be useful for novice programmers. This is an intuitive result, as many bugs can arise from an incomplete or incorrect understanding of the programming language, and only require completing or correcting the appropriate knowledge before the novice is able to correct the bug. Unfortunately, this area lacks new developments for the C programming language. This may be due to the difficulties of developing tools for the C programming language: the lack of standard methods for parsing and semantic analysis, and the complexity of the language.

Our explanatory system is built upon the Clang libraries, providing robust and sustainable parsing and semantic analysis of the C programming language. It is designed to operate independently of the SeeC system, so that it may be reused in other Clang-based educational tools. The system creates natural language explanations for individual nodes in Clang’s Abstract Syntax Trees. To illustrate the implementation of our system, consider the small piece of code in Listing 2.

Listing 2 Example function

```
1  int isodd(int n) {
2     if (n % 2)
3         return 1;
4     else
5         return 0;
6 }
```

For this example function, Clang produces the AST that is represented by Figure 3. Clang’s node class hierarchy has two distinct base classes: \texttt{Decl} for declarations, and \texttt{Stmt} for statements. Each node contains detailed semantic information, as well as precise locations for the node’s representation in the source code.

The interface to the explanatory system is designed to be as simple as possible. Clients pass in an AST node, and the system returns either an explanation for the node, or an error describing the reason that the explanation could not be generated.

Each node class provides access to specific information for the particular kind of declaration or statement that it represents. The hierarchy also contains abstract classes that provide access to information that is shared by multiple kinds of nodes. For example, the \texttt{FunctionDecl} in Figure 3 is a subclass of \texttt{NamedDecl}, which allows us to retrieve the name of a node (for this node it is “\texttt{isodd}”, the name of the function). Our system uses this information to tailor explanations to the specific nodes that are being explained, rather than using a fixed explanation for each kind of node.

The system is also designed to be fully internationalized, for which we use the International Components for Unicode (ICU) system\(^4\). Explanation text is stored in an ICU resource bundle, containing a unique entry for each kind of declaration and statement. After the text is retrieved it is formatted using ICU’s message formatting system, and provided with information that we have collected from the AST node. As an example, let us consider the generation of an explanation for the \texttt{IfStmt} in Figure 3. The following information will be collected from the node:

- \texttt{has\_condition\_variable} Whether or not the if statement’s condition contains a variable declaration. In this case the value is “false”.
- \texttt{has\_else} Whether or not the if statement has an else branch. In this case the value is “true”.

The explanation text then uses the ICU message formatting system to vary the generating explanations based on this value. For example, the explanation for an \texttt{IfStmt} may contain the following:

\[
\{ \texttt{has\_else}, \texttt{select}, \texttt{true} \} \{ \texttt{It consists of a condition, a body, and an else.} \} \texttt{false} \{ \texttt{It consists of a condition and a body} \}
\]

For our if statement’s node the value of \texttt{has\_else} was “true”, so this part of the explanation will be formatted into the text “\textit{It consists of a condition, a body, and an else.}\textsuperscript{\(5\)}”. Explanations often refer to other nodes in the AST, which may be child nodes that are contained in a subsection of the explained node’s source code, or may be in an altogether different location. In our example above three AST nodes are referenced: the if statement’s “Condition” is a \texttt{BinaryOperator}, its “Body” is a \texttt{ReturnStmt}, and its “Else” is also a \texttt{ReturnStmt} (as we can see in Figure 3).

We developed a simple system to explicitly embed this referencing information into the explanatory text. Each kind of node can provide a dictionary of related AST nodes. Our example \texttt{IfStmt} provides three: “\texttt{cond}” for the condition, “\texttt{then}” for the body, and “\texttt{else}” for the else. The explanation text is modified to reference these dictionary entries as follows:

```
It consists of a \texttt{@[cond]condition@[]}, a \texttt{@[then]body@[]}, and an \texttt{@[else]else@[]}.
```

\(^4\)http://site.icu-project.org
The explanation that is returned from the system contains, as well as the formatted text, information about the areas of text that are linked to AST nodes. The display explanation that we integrated into SeeC’s trace viewer uses this information to highlight related AST nodes when the student’s mouse cursor hovers over a section of the explanation text. This allows novice programmers to quickly check which area of the code is referred to by the explanation, receiving instant visual feedback. A reference can also use a URL rather than a related node’s key, providing the ability to link explanatory text to external material. For example, we use this to link explanations to appropriate lecture notes.

The system can optionally use information about the runtime state of the program when generating explanations. This information is provided to the system using callback functions which receive statement nodes and return information about the value produced by the statement: whether or not it exists, a string describing its value, and if possible an implicit conversion of the value to a bool. This information is provided to the message formatting system in the same manner as the semantic information provided by the AST nodes. To return to our example, the explanation of if statements can explain whether the body or the else statement is executed based on the value that was produced by the condition statement.

6 Integration into SeeC

The systems that we have introduced were developed as discrete components, with the aim of fostering reuse and extension. However, we also designed them for use by students in a simple, unified system. We have integrated the graphical visualization system and explanation generation system into SeeC’s graphical trace viewer (Figure 4).

The SeeC system, described in Section 3, uses compile time instrumentation to automatically detect runtime errors during the execution of student programs, and to record the execution of student programs into trace files. The graphical trace viewer loads these trace files, allowing students to inspect the recorded state of the program at any point during its execution. Students may navigate forwards and backwards through the execution trace using the simple controls at the top of the viewer.

The system also supports contextual navigation based on particular items in the state. For example, students may select a particular value in memory and then navigate to the allocation of that memory, the most recently occurring write to that memory, or the eventual deallocation of the memory. A student may also select a particular function call and rewind to the beginning of the call or move forwards until the call is complete. These features have been integrated into the display of the graphical visualization of process states, allowing students to navigate by interacting with values or nodes in the graph.

7 Summary and Future Work

We have discussed the design and implementation of robust, maintainable, reusable systems for visualizing the runtime memory state of students’ C language programs, and for generating natural language explanations of those programs. These systems have been integrated into SeeC’s graphical trace viewer, augmenting SeeC’s existing novice-focused debugging features. Where previous tools for the C programming language have relied on custom written parsers and interpreters, our systems are built upon the Clang libraries which provide high quality language support and are being constantly improved and maintained by a strong, active community.

One of the problems with visualizing the memory state of C language programs is the task of determining which of multiple competing types should be rendered for a particular area of memory. Currently we render all possible interpretations of a value, but some modification to the underlying system may we able to record which member is used when storing a value into a union, and then use this information to render only the “active” member of the union. We also deal with ambiguous memory caused by type-punned pointer aliasing. We try to reduce this by selecting a single type to render, and allowing students to override this with their own selection, but in some cases it may be useful to render multiple competing types, using some visual cue to indicate that they occupy the same space in memory. This concept may be difficult for novice programmers to understand, so one would have to carefully evaluate the visualizations to determine whether they presented useful information or further confused the students.

SeeC’s instrumentation checks for many runtime errors. If an error is detected then it is recorded in the execution trace, and it will be visible in the states recreated from the execution trace. The trace viewer currently displays runtime errors using the natural language descriptions that are generated by the underlying system, but some errors could also be displayed by the visualization system. For example the error that is described in Figure 4 (displayed in-line in the source code) is raised when a function expects a C string but is passed a pointer to a character array that is not null terminated. The visualization system could highlight the referenced character array and illustrate that there is no terminating null character.

Generating explanations based on AST nodes is a practical method that allows us to leverage the Clang libraries to provide robust and detailed explanations of students’ programs. However, even relatively simple statements in the C programming language may consist of several AST nodes. A student considering an entire statement must view the explanations for the individual AST nodes. It may be possible to create a system which can combine fragments of explanations to create a unified explanation for an entire statement, without losing the internationalization of our current system. A brief fragment describing a node could link to a detailed, node-specific description such as those generated by our current system.

Any educational system must naturally be evaluated to determine its merit, though we are hopeful that our systems will prove as beneficial to students as the prior systems that influenced them. In the 2nd-semester 2013 presentation of our first year course on Operating Systems and the C Programming Language we will employ the complete system described here, including the graphical visualizations and natural language explanations. During this time we will investigate students’ usage of the system to determine whether or not they find individual components useful, and to evaluate how students use those features to debug their programs and to increase their understanding of the programming language.

Lahtinen (2009) argued: “If we want visualizations to catch on in mainstream CS education, we need to study their usage in realistic learning situations in real CS class rooms and adapt the visualizations to suit these conditions.” In following these guidelines,
as well as many similar recommendations, we aim to study students’ use of SeeC during their regular coursework. We plan to perform several evaluations using various approaches in order to construct a more complete picture of the system’s use. One method that we intend to employ is to record students’ interactions with the graphical trace viewer, allowing us to investigate how students use the system during the normal course of their studies and without the interference of a human observer.

The complete SeeC system is free and open source, including the additional components that we have introduced. Interested readers are invited to contact the authors to discuss the tool’s suitability for their courses.

References


Abstract

Most instructors teaching Computer Science use examples to help students learn, and many instructors use worked examples (either in a static or a dynamic style) in their courses. However, the research on worked examples is not well known in the Computer Science Education community. This paper provides an overview of how worked examples have been studied, and the major findings from the literature, particularly as they relate to Computer Science.

Keywords: cognition, learning, cognitive load theory, worked examples

1 Introduction

Shulman (2005) uses the term signature pedagogies to describe pedagogical practice that is characteristic of a given discipline. These are the ways of teaching that spring to mind when we think of a particular discipline — for Medicine, it is the bedside teaching that occurs during clinical rounds where groups of students are involved in discussions with a resident; for Law, it is the case dialogue method in which a complex case is dissected through discussion and argument. We believe that the use of worked examples to demonstrate problem solving and software development is a signature pedagogy for Computer Science. Yet this key pedagogical practice, characteristic of education in Computer Science, has not been widely studied in the very context of Computer Science.

According to Atkinson et al. (2003) “Worked-out examples typically consist of a problem formulation, solution steps, and the final answer itself”. A problem is presented, accompanied with step-by-step instructions which lead to the solution. These are usually textual but may include pictures, diagrams or animations. We consider that this definition of worked examples would include dynamic demonstrations of problem solving (such as live demonstrations of writing programs that solve simple problems). Students are expected to study the worked example and from it learn how they might apply it to similar problems. Figure 1 illustrates a typical worked example in Computer Science.

According to Miller (1956), humans have a limited working memory, where only a few chunks of information can be processed at one time. Cognitive load (Sweller 1988, Chandler & Sweller 1991) describes the amount of information that must be held in working memory during the process of problem solving. If the working memory is overtaxed, for example, by trying to solve a problem without enough scaffolding, learning performance will suffer. It is for this reason that Kirschner et al. (2006) argue that problem solving fails to be an effective learning strategy when there is insufficient guidance in place.

Humans also have a long-term memory with a much larger capacity (Baddeley & Hitch 1974). Long-term memory consists of a set of schemas, and with practice, information stored according to the schemas can automatically be recalled and applied with minimal impact on working memory. In this model of human cognition, the aim of teaching is to help students form appropriate schemas, which can in turn be used to solve both familiar and novel problems.

Recent literature distinguishes between different types of problem solving strategies. For example, in the domain of software development, working examples can serve as a form of guided practice, where students are provided with a step-by-step demonstration of how to solve a problem. This can be particularly useful when students are first learning a new programming language or software tool. Working examples can help students build a working memory for the new material and form a set of schemas that they can apply to solve similar problems in the future.

Problem statement:

Write a function that calculates the area of a rectangle.

Solution Design:

1. Determine what parameter(s) the function needs to calculate an answer, as well as their type(s):
   - width (float), height (float)

2. Determine what result the function will return, including the type:
   - the area of the rectangle (float)

3. Determine the steps needed to calculate this result:
   - To calculate the area of a rectangle we will use the formula:
     \[ \text{area of rectangle} = \text{width} \times \text{height} \]

Implementation:

1. Using the identified parameters, write the function header:
   ```python
   def rectangle_area(width, height):
   ```

2. Using the identified steps, calculate the result:
   ```python
   def rectangle_area(width, height):
       area = width * height
   ```

3. Return the final result:
   ```python
   def rectangle_area(width, height):
       area = width * height
       return area
   ```

Figure 1: An exemplar worked example
of cognitive loads — *intrinsic* cognitive load, *extraneous* cognitive load and *germane* cognitive load (Paas et al. 2004).

**Intrinsic cognitive load** is imposed by the degree of interaction between elements in the problem domain. It may not be reduced unless the content is in turn reduced, and is therefore unaffected by altering the presentation of material by an instructor.

**Extraneous cognitive load** is caused by activities which do not assist with the formation of schemas. These activities interfere with learning because they require the use of working memory for processes that are not related to the focus of learning.

**Germane cognitive load** relates to the higher level processes (scaffolding) that supports the formation of schemas, and therefore improve the effectiveness of the activity for learning.

For further information, work by Caspersen & Bennedsen (2007), and Caspersen (2007) provide excellent overviews of the theory of cognitive load theory as it applies in the practice of instructional design for Computer Science.

Although examples, and in particular, worked examples are widely used to teach Computer Science, there are few studies that have investigated their effectiveness in the Computer Science context. In this paper we present the theoretical basis and research findings for worked examples, which may encourage practitioners to be more deliberate about the organization of their own examples. We also show how worked examples have been studied in related fields like engineering and statistics, and examine the literature to identify potential avenues for further research in Computer Science.

2 Ways of presenting worked examples

We first consider the different ways that worked examples can be integrated into the overall instructional design for a given topic.

**Examples only**: In this approach, students are simply provided with a set of worked examples. There are no activities, such as exercises or problems to solve, associated with the examples.

**Example-problem blocks** This approach provides students with a block of worked examples of various types to study, then a set of related problems are given which students are expected to solve.

**Example-problem pairs** These are one of the most common ways of presenting worked examples, where each example is paired with a problem similar to the example for students to complete. Students alternate between studying a worked example and solving a related problem.

**Faded worked examples** In this approach, a complete worked example is presented, then another worked example with one step missing is presented, and students are expected to fill in the missing step. They are presented with a series of worked examples, with an extra step removed each time, until a student is presented with just a problem to solve.

The most common orders for fading steps are known as forward fading - where steps are removed starting from the beginning, and backwards fading - where steps are removed from the end first.

2.1 Other techniques that support worked examples

These basic forms of presenting worked examples are often augmented with other techniques, such as:

**Subgoal labeling** A technique where groups of steps are given a label, to help organize the information into a meaningful structure. According to Margulieux et al. (2012) subgoal labels allow students to focus on groups of steps rather than individual steps, giving them fewer problem-solving steps to consider and, reducing cognitive load.

The highlighted structure given by subgoals is also supposed to assist with schema formation, or provide “mental model frameworks” to internally explain how problems are solved.

**Self explanation prompts** Self-explanation is a process some learners undergo when provided with a worked example. Students who try to explain to themselves the reasons for a step or set of steps in an example were found to learn more than those who don’t (Atkinson et al. 2003), so self-explanation prompts are designed to elicit such self-explanations. Self-explanation prompts can be in the form of asking students to justify a step or choosing what principle a particular step is invoking. When employed correctly, these prompts are considered to be a source of germane cognitive load.

3 What is the effectiveness of Worked Examples?

The benchmark for evaluating worked examples is usually some form of problem solving task. The task typically requires a student to solve a problem, and the student is told when their solution is correct. Usually a set of questions is given, and some of these questions are swapped for worked examples — people in the problem solving condition solve all the questions, and people in the examples condition study several examples and solve some problems.

They are also often evaluated for their ability to promote near transfer and far transfer. Near transfer is the ability of students to solve questions which are isomorphic to the ones they saw in their training phase, whereas far transfer is the ability for students to solve novel problems which use many of the same skills from the training phase, but in a different sequence or with some of the learned techniques requiring minor modifications.

3.1 Examples only

The provision of examples over giving problems to solve reduces extrinsic cognitive load and directs student’s attention to the relationships between different problem steps, thereby encouraging students to construct relevant problem-solving schemas around it. Problem-solving with no guidance, however, requires a large cognitive load for novices, but all the effort goes to finding an answer rather than schema formation.

Studies have investigated the use of isolated worked examples to illustrate how to solve a given problem in fields such as Accounting (Stark et al. 2002), Electrical Engineering (van Gog et al. 2006), and CNC Programming (Paas et al. 2004).

In the domain of CNC programming, Paas et al. (2004) found that presenting multiple worked examples with high variability resulted in improved learning compared with multiple worked examples with low variability. They also compared worked examples only with a problem-example pair condition, and found that attempting to solve a problem prior to the worked example actually impeded learning.
A later study by van Gog et al. (2011) compared worked examples on their own, example problem pairs, and worked example-pairs and problem solving on its own for teaching high school students to diagnose a faulty electrical circuit.

The use of worked examples resulted in improved learning and transfer compared with traditional problem-solving techniques. This improvement in learning was also observed in the condition where students were presented with example-problem pairs. Students reported lower metacognition and scored better results upon testing than those in the problem solving condition, or the problem-example paired condition. No difference was found between example-only and example-problem pairs van Gog et al. (2011).

Although it might seem that presenting a problem first would motivate a student to engage more deeply with the worked example, the results of these studies suggest that greatest learning occurs if the worked examples are presented prior to the problem.

### 3.2 Example-problem blocks

The use of example-problem blocks is uncommon, but has been studied in a programming context. In one notable study, Gregory et al. (1993) compared using example-problem blocks, example problem pairs, alternating similar problem-solving task, and blocks of problem-solving tasks.

The tasks were 6 pairs of LISP programming questions, to solve after having gained some familiarity with LISP before the experiment proper started. Each pair tested the same skills, with one being the source problem and the other being the target. The idea was that the source provided a chance to initially learn to solve the problem, and the target allowed them to practice the techniques learned from the source.

For the example-problem pairs and block conditions source problems were swapped for a worked example. In the block conditions, sources were separated from targets whereas in the pair conditions targets immediately followed sources. In other words, the example-problem paired condition involved a sequence of problems where each problem was preceded with a worked example. The block condition involved a sequence of worked examples, followed by a sequence of problems to solve.

Example-problem blocks were the worst preforming group in post-tests. Students in this condition spent as much time studying source examples as the example-pair group, but spent more time on the target problems. Gregory et al. (1993) suggest that difficulty in remembering the examples once they met the equivalent problem would hinder later problem solving, and that if students are unable to recall the appropriate example, the benefit of studying them over problem solving disappears.

Indeed, both of the problem-solving groups performed better than the example-problem blocks group, suggesting the extra practice afforded to the problem-solving block group outweighed the benefits of having worked examples. The example-problem pairs were the best performing group on post-tests.

### 3.3 Example-problem pairs

Extensive work by Sweller and his colleagues has established that worked examples, when paired with problems, are superior to problem-solving without worked examples in a variety of subject areas (Sweller & Cooper 1985, Mwangi & Sweller 1998). Fewer studies have compared the use of example-problem pairs with other configurations of example and problem presentation.

The use of example-problem pairs is thought to foster learning better than example–problem blocks, as students can better select and recall (if they just studied) to relate the problem to when they are given one directly after the other. Separating them may make it harder to recall the relevant example to relate to the current problem.

As described previously in section 3.1, van Gog et al. (2011) compared worked examples on their own, example problem pairs, problem example pairs and problem solving, and found that the use of example pairs to work more effectively than the other conditions. Example-only and example pairs performed similarly.

When example-problem pairs are compared with example problem blocks, results suggest that the example-problems pairs are effective for learning Gregory et al. (1993). Students studying examples in both paired and block conditions spent equal time, but those who were given problems to solve immediately following the examples appeared to be able to solve later problems more efficiently than those students who studied a block of examples prior to practicing the problem solving skills.

Renkl et al. (2002) conducted three experiments comparing backward and forward fading with example-problem pairs. The first experiment compared the effectiveness of backwards fading with example problem pairs for solving Statistics problems. The second experiment compared forward fading with example-problem pairs in the context of Physics. The third experiment compared both forward and backward fading with example-problem pairs. In all three cases, students in the fading conditions outperformed those using example-problem pairs for near transfer problems. Students also produced fewer errors during learning. This suggests fading may offer better learning outcomes in a shorter amount of time for near-transfer tasks than example-problem pairs.

Atkinson et al. (2003) explores the use of backwards fading compared to example-problem pairs for solving statistics problems. Under a variety of conditions, backwards fading resulted in higher post-test results than example-problem pairs on both near and far transfer problems.

### 3.4 Faded worked examples

Although worked examples appear to be more effective than simple problem solving under a variety of conditions, as a student gains expertise from studying worked examples, the benefits of studying them over problem solving disappears (Renkl et al. 2002, Atkinson et al. 2003). It is thought that partial schema formation means that the elements that were once a source of germane cognitive load become a source of extrinsic cognitive load. At this point problem-solving without worked examples becomes more effective (Renkl et al. 2004, Kalyuga et al. 2003).

To ease this transition, faded worked examples begin with a fully worked example, but as they study it and gain expertise steps are removed to encourage a manageable amount of problem solving, fostering germane cognitive load. By the end of the fading sequence, students have studied many of the worked example steps and will be able to problem solve on their own.

Studies investigating the sequence of fading have not produced reliable findings. Renkl et al. (2002) suggested that backwards fading worked would produce better results than forward fading on near transfer items. However, later work was unable to confirm these findings. Renkl et al. (2004) investigated whether the sequence of fading affected near and far transfer more than the types of steps removed. In their two experiments no difference was found in learning outcomes or errors during learning.

Their results also suggested that students learn most
about those steps which are faded. The implication is that the learning activities elicited by removing steps focuses students on the area. For this reason, they suggest the earlier results must be attributed to the learning material they used and the type of the steps removed. The backward procedure removed those steps that may be 'pre-requisites' or otherwise helped students learn principles which were helpful for earlier steps. Doing it the other way means they would not learn the important principles first, which would hinder subsequent learning.

Moreno et al. (2006) also compares forward fading and backward fading. Those who used forward fading were found to outperform those using backwards fading. They suggest this has to do with the ease of the material they were learning. Having studied the first example, students may have gained all the initial knowledge they needed.

According to the expertise reversal effect, if a student already has some expertise in the area, further learning is better gained by problem-solving, and techniques like worked examples may hinder or decrease subsequent performance. This is because for an expert, studying a worked example is a source of extraneous cognitive load rather than germane cognitive load, and problem-solving promotes germane cognitive load for those with some expertise in the targeted domain. Because forward-fading gets students to start problem-solving as the first, rather than the last step, the early problem-solving may have benefited students as opposed to those who had to wait until the final step to problem-solve.

3.5 Self-explanation prompts

Atkinson et al. (2003) cites research with mixed results on the effects of activities designed to elicit self explanations. It has been suggested that self-explanations are a source of germane cognitive load, helping students to form schemas around the materials they’re learning, rather than e.g. just memorizing a set of steps to a solution. Experiments where students were prompted by an online tool to fill in templates for self-explanations, or where students were encouraged to write their own self-explanations as comments, failed to increase learning gains consistently. Another study found self-explanation prompts during the problem solving phase rather than during example study received positive results on learning.

In their own study into solving statistics problems, Atkinson et al. (2003) prompted students with a set of principles a given step in the worked examples may be drawing from. Students were expected to choose one of the principles, and this was expected to foster self-explanations. Students in the self-explanation groups performed better on post-tests for near and far transfer problem than those not prompted in the equivalent fading or example-pair groups not prompter. No extra time was required to achieve this result. The results for self-explanation prompts with backwards fading were replicated for university and high school students.

3.6 Subgoal labeling

Margulieux et al. (2012) studied the use of subgoal labelling in video demonstrations and instructional material for creating mobile applications. In the subgoal conditions, the steps in the demonstration video and instructional material were labelled with subgoals grouping several steps into a cohesive group.

In post-tests participants in the subgoal group better identified subgoals necessary to complete a solution whether or not they complete it correctly or not. They also were more likely to correctly complete the subgoals necessary for the assessments. Overall the subgoal condition outperformed their counterpart on both assessments immediately after training and assessments one week later. They did so spending less time on the assessments, and were less likely to drag out blocks in the assessments.

3.7 The expertise reversal effect

Although studies of worked examples generally shows positive benefits for learners, Kalyuga et al. (2003) demonstrate instances where providing worked examples can hinder learning. For novices, worked examples direct their attention to important features of the problem and help in forming relevant problem-solving schemas. This is a better use of their cognitive resources than problem solving, which requires extensive search of the problems space (Sweller 1988). Unguided problem solving imposes a heavy cognitive load unrelated to schema formation. In other words, it is a source of extraneous cognitive load, but not germane cognitive load.

However someone with some expertise already has partial or full schemas in long-term memory. For experts, worked examples are redundant. The effort required to analyse worked examples becomes a source of extrinsic cognitive load rather than germane cognitive load. Kalyuga et al. (2003) identify studies involving trades apprentices, students working with databases and other experiments where people with more experience fail to gain any benefit from worked examples. In these studies, as novices’ expertise increases, they learn more from problem solving rather than studying examples.

4 Examples in Computer Science

There is little research into worked examples in Computer Science. Early research into cognitive load theory drew upon work in teaching LISP (e.g. Anderson et al. (1984)), where it was observed that students would rely heavily on provided examples as opposed to instructional texts. Much of the worked example literature rely on the results of these studies, but nonetheless worked examples have not been well studied in Computer Science education, as Merrinboer & Paas (1990) and Mason & Cooper (2012) observed.

A few reports in the CS Education literature focusing on the instructional design of introductory programming courses have advocated the use of worked examples during the course (Hsiao et al. 2013, Caspersen & Bernedsen 2007, Lui et al. 2008, Gray et al. 2007). However, formal studies of worked examples in the context of Computer Science, such as that of Gregory et al. (1993) and Margulieux et al. (2012) are the exception rather than the rule.

4.1 Faded Worked Examples in Computer Science

Gray et al. (2007) provides a detailed discussion of how faded worked examples might be applied in an introductory programming course in Computer Science. We examine their approach in this section. The task of programming is decomposed into components whose cognitive load they claim can be adequately managed. The decomposition is based on two parts: the abstract algorithmic dimensions and the associated concrete programming constructs. The algorithmic dimensions identified are design, implementation and semantics (the meaning of supplied code). The semantic dimension is divided in three, into assertion (students should be able to state true statements about the code at various point of execution), execution (given an input, provide the output) and verification (be able to test the code). The programming constructs chosen were selection, iteration and subroutine calls. Each of these would be taught in pairs (design of a selection
algorithm, implementation of an iterative algorithm etc.), with the learning of each pair supported by sets of faded worked examples.

Concrete, fully worked examples are provided for all of the design-construct and implementation-construct pairs, and provide an example of semantic-assert and semantic for selection algorithm. Although it is useful for instructors considering adopting this pedagogy to have such examples, they have not been used in any formal studies or actual courses.

Although Gray et al. (2007) suggest the use of backwards fading, Renkl et al. (2004) suggests that the success of backwards fading compared to forward fading is an artefact of the teaching materials people use rather than something inherent in the backwards sequencing. The sequencing of fading should be examined to see which steps may be prerequisites for understanding other steps — Renkl et al. (2004) suggests these kinds of steps should be removed first.

The use of ‘ASSERT’ during the semantic part of training is designed to get students to state what is known about certain parts of code in the form of code comment. This is motivated by the same principals motivating the use of self-explanation prompts in Atkinson et al. (2003). However, it is not clear how students will learn how to develop their own assertions without scaffolding. An explicit process to help students develop assertions is provided for the selection statements, but no such process is provided for other syntax constructs.

As mentioned earlier, the research on self-explanation prompts is not unanimous. Atkinson et al. (2003) suggests the interface allowing students to write down self-explanations may have an effect on whether it will be effective, and the prompts they provide in their own experiments require students to make choices from a list, rather than generating them on their own. This requires a low amount of activity from students. The scaffolding provided means they won’t have to come up with assertions from scratch like in some previous studies, but the suggested ‘ASSERTS’ require a little more than picking options from a list. Further study on the use of self-explanation prompts, or assertions during code development, is required, both theoretical and empirical.

However, all in all, Gray et al. (2007) provide a clear framework for using and testing faded worked examples in Computer Science. Such techniques could straightforwardly be extended to other C derived languages like C, Java or C2, or any kind of imperative or procedural language. Other constructs or dimension of programming could be considered too.

5 Implications for Computer Science

The use of examples is extremely common in the discipline of Computer Science, particularly in courses that introduce programming concepts. It is fairly typical in lectures, and in most textbooks, for numerous examples of code to be shown to students. These examples frequently take the form of code traces, where the instructor presents some code and proceeds to demonstrate how it would be executed by tracing the execution one step at a time; and problem solution pairs, in which a problem is posed by the instructor (e.g., “Write a method that determines whether a given number is a prime number or not”), and a solution is subsequently presented and the code is explained in detail.

Less commonly, instructors may demonstrate the development of software by programming in real time during the lecture.

However, it is far less common for students to engage in problem solving activities during lecture time. Certainly, reports of active learning in the Computer Science classroom illustrate how such activities are possible, but these are not widespread in practice. In most courses, it is only much later, during homework or in laboratory sessions, that students solve problems similar to those covered during lectures. In other words, most courses use the instructional design of example-problem blocks. Although the use of example-problem blocks has not been extensively studied, there is some indication that it is one of the least effective approaches Gregory et al. (1993).

It is possible that some of the difficulties observed in the novice programming literature may be due to intrinsic cognitive load imposed by the complexity of programming tasks. If, as claimed by Sweller & Chandler (1994), programming is an intrinsically difficult area, then it is extremely important to minimize the extraneous cognitive load if students are to be successful. Although the studies presented here suggest that some ways of organising worked examples are more effective than others, more research on the cognitive load imposed by programming is required to better understand how to organise and present content in the most effective way.

Additionally, it may be beneficial for practitioners to reflect on the organisation of their course material in the light of the studies discussed here. Some simple changes in the way examples and exercises are structured could improve learning for students in most programming courses.

6 Conclusions

The evidence suggests certain worked example techniques (primarily example-problem pairs and faded worked examples) are an improvement over standard problem solving techniques, in terms of learning time and performance on near transfer tests in novices.

In situations where the student is not a novice, faded worked examples appear to improve performance and decrease learning time on near transfer tasks. In addition, techniques such as self-explanation prompts may promote far transfer as well if applied appropriately.

Since much of the research involves well structured domains like Statistics, Physics and Engineering, it is likely that findings would transfer readily to the domain of Computer Science. However, further studies are required to confirm the effectiveness of pedagogies based on worked examples in the context of Computer Science. The use of faded worked examples with self-explanations has the potential to help students learn more effectively, but the best order of fading problems is currently unknown Renkl et al. (2004). Future research into what steps should be faded first for a given problem in Computer Science would also help us understand how faded worked examples could most effectively be employed.

References


Caspersen, M. E. & Bennedsen, J. (2007), Instructional design of a programming course: a learning theoretic


Miller, G. (1956), ‘The magical number seven, plus or minus two: Some limits on our capacity for processing information.’, *Psychological Review* 63, 81–97.


Manifestations of Preoperational Reasoning on Similar Programming Tasks

Donna Teague
Queensland University of Technology,
Brisbane, QLD, Australia
Tel: +61 7 3138 2000
d.teague@qut.edu.au

Raymond Lister
University of Technology, Sydney,
Sydney, NSW, Australia
Tel: +61 2 9514 1850
Raymond.Lister@uts.edu.au

Abstract
In this research paper, we study a simple programming problem that only requires knowledge of variables and assignment statements, and yet we found that some early novice programmers had difficulty solving the problem. We also present data from think aloud studies which demonstrate the nature of those difficulties. We interpret our data within a neo-Piagetian framework which describes cognitive developmental stages through which students pass as they learn to program. We describe in detail think aloud sessions with novices who reason at the neo-Piagetian preoperational level. Those students exhibit two problems. First, they focus on very small parts of the code and lose sight of the "big picture". Second, they are prone to focus on superficial aspects of the task that are not functionally central to the solution. It is not until the transition into the concrete operational stage that decentration of focus occurs, and they have the cognitive ability to reason about abstract quantities that are conserved, and are equipped to adapt skills to closely related tasks. Our results, and the neo-Piagetian framework on which they are based, suggest that changes are necessary in teaching practice to better support novices who have not reached the concrete operational stage.

Keywords: Neo-Piagetian theory, novice programming, think aloud.

1 Introduction
It is a common source of frustration for computer science educators that novices do not transfer to a second programming problem the concepts taught on an initial problem. For example, we posed to novice programmers the tasks shown in Figures 1 and 2. We found that some students who could do the first task could not do the second task. We posed these questions to two classes, in different semesters. Table 1 shows the performance of both classes on the second task. In both semesters, the percentage of students who answered the second task incorrectly was worse than we expected, given the number of weeks of instruction the students had received.

![Figure 1: The shift task with an explicit temp variable](image1)

![Figure 2: The second shift task](image2)

**Table 1: Performance on the Task in Figure 2**

<table>
<thead>
<tr>
<th>Week of Semester</th>
<th>No. of Students</th>
<th>Percentage wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>51</td>
<td>37%</td>
</tr>
<tr>
<td>6</td>
<td>113</td>
<td>53%</td>
</tr>
</tbody>
</table>

To understand why so many students struggled with such a simple task, we began the qualitative research study described in this paper. In our study, we had 11 volunteer students complete the tasks in Figure 1 and 2, while having those students think aloud as they did so. Table 2 summarises the performance of the 11 students. The names shown in that table are all
pseudonyms. All of these students were in at least their third week of learning to program. All 11 students completed the first task successfully. In completing that first task, those 11 students demonstrated that they understood assignment statements, and that they understood the English language instructions associated with both tasks. However, 3 of the 11 students could not then solve the second task, and a fourth student (Jim) took much longer. (Those four students are in the shaded region of Table 2.) This brings us to the research question addressed in this paper:

**Research Question:** Why can some students answer correctly only one of the two problems shown in Figures 1 and 2, when both tasks require functionally identical code?

Note that our research question is not related to the prevalence of this issue in the general population of programming novices. Given the small group of students we studied, and that those students are from a single institution, it would not be appropriate to speculate on prevalence. However, what we can do in a qualitative study of this type is arrive at a possible explanation for why some students find the second task to be significantly harder than the first task. The type of microgenetic analysis that we carry out in this study has been applied in many domains to test theories of cognitive development (Siegler 2006) and has also been used before in a study of novice programmers (Lewis 2012).

We were able to make sense of our research data via neo-Piagetian theory. In the next section, we briefly describe that theory. We then present our transcript data from three students, two of whom struggled on the second task while the third student was able to do both problems quickly. We interpret that transcript data using the neo-Piagetian theoretical framework.

2 The Neo-Piagetian Stages

Lister (2011) proposed, in accordance with neo-Piagetian theory, that there are four main stages of cognitive development in the novice programmer. At the least mature stage, the sensorimotor stage, a novice programmer cannot reliably trace a given piece of code (i.e., manually execute it). The sensorimotor approach to writing a trace on paper is ad hoc and often inconsistent. Also, they commonly have misconceptions about what various programming constructs do (Du Boulay 1989). Furthermore, these novices often apply a misconception at some points in a trace and then apply a correct conception at other times.

The next neo-Piagetian stage is preoperational. Novices at this stage can trace code accurately, but they struggle to reason about code. That is, they have difficulty understanding how several lines of code work together to perform a computation. At any point in time, these novices tend to be focused on small parts of the code, and ignore the implications of code they have already considered. This is what neo-Piagetian theorists refer to as spatial and temporal centration.

At the concrete operational stage, novices can reason with abstractions of code (e.g., diagrams). They can also reason about the concept of conservation which Flavell (1977) describes as “… a quantitative invariant among transformations”. We elaborate on the concept of conservation in the following sub-section.

Finally, there is the formal operational stage, which is the stage educators hope their students will reach. At this stage, novices can reliably and efficiently “problem-solve”; they understand and use abstractions, form hypotheses and can make inductive and deductive inferences.

By analysing students' answers in an end-of-semester exam, Corney et al. (2012) provided indirect evidence that novices pass through some of these neo-Piagetian stages. However, such evidence does not provide a direct indication of the actual thought processes of students. Think aloud studies have also been undertaken with students who were given programming code to hand trace and explain in plain English (Teague, Corney, Ahadi, and Lister 2013). The results provided evidence of preoperational reasoning by some of the students.

In this paper we provide direct empirical evidence of students' thought processes while solving code writing tasks, specifically the tasks shown in Figure 1 and 2.

2.1 The Concept of Conservation

According to neo-Piagetian theory, it is only at the concrete operational stage that a novice has developed the ability to reason reliably about abstract quantities that are conserved, and the novice is not deceived by superficial appearances. For example, Flavell (1977) describes an experiment where a preoperational child believes that when clay is moulded into different shapes the amount of clay changes. A child at the concrete operational stage is not deceived by such perceptions. Lister (2011) proposed that in a programming context, a novice at the concrete operational stage should be able to easily make minor changes to code while conserving what the code achieves, while the preoperational novice programmer would struggle to do the same. The contribution of this paper is providing empirical evidence for that proposal.

Our objective was to see if any of our novices demonstrated an ability to conserve a specification when given a small change to the implementation. Specifically, we wanted to see if any of our novices could solve either the first or second task, but not both. Our hypothesis was that students who are operating at the preoperational level will struggle to apply consistently the abstract principal common to both problems – that saving a variable to temp makes it possible to overwrite that value in the copied variable. In neo-Piagetian terminology, this abstraction is referred to as the "invariant amid transformations" (Flavell 1977).

2.2 Working with Cyclic Series

Our two programming tasks are analogous to an experiment Piaget conducted where he asked children to predict the next element in a cyclic series (Piaget 1971a). To do so required the children to translate the elements into a linear series. Piaget found that relationships of order are operational. That is, people are not capable of dealing with such a concept until the concrete stage.
At the sensorimotor stage, people are barely able to manage translating a cyclic series into a linear series and unable to foresee successive elements. At the preoperational stage people have the ability to predict successive elements in a cyclic series if they start at the first element. Towards the end of the preoperational stage, people can cope with intermediate starting points, but still fail to predict elements beyond the last.

Our programming tasks involved transforming a cyclic series (the diagram) into a linear series of assignment statements to achieve a ‘movement’ of values.

3 Think Aloud Results
At some point in time after performing a think aloud on the first task, the 11 students performed a think aloud on the second task. The elapsed time between think alouds varied from student to student. Table 2 provides the specific information for each student.

Table 2 shows the total time taken to complete (or abandon) each task. The data in Table 2 is sorted by length of time spent on the second task. Thus the four students at the bottom of Table 2 (i.e. in the more heavily shaded section of the table) took the longest time to complete the second task. According to the arguments we have made above, those four students are likely to be at the preoperational stage of development.

Table 2 also shows the level of assistance provided to each student by the person conducting the think aloud.

Table 2: Think Aloud Performance on Shift Tasks

<table>
<thead>
<tr>
<th>Alias</th>
<th>The First Shift Task (see Figure 1)</th>
<th>The Second Shift Task (see Figure 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (minutes:seconds)</td>
<td>Help Given</td>
</tr>
<tr>
<td>John</td>
<td>2:03</td>
<td>0. none</td>
</tr>
<tr>
<td>Steve</td>
<td>1:48</td>
<td>1. clarify</td>
</tr>
<tr>
<td>Becki</td>
<td>1:05</td>
<td>0. none</td>
</tr>
<tr>
<td>Michael</td>
<td>1:24</td>
<td>1. clarify</td>
</tr>
<tr>
<td>Bobcat</td>
<td>14:36</td>
<td>3. hint</td>
</tr>
<tr>
<td>Lance</td>
<td>3:10</td>
<td>0. none</td>
</tr>
<tr>
<td>Johnstone</td>
<td>4:48</td>
<td>3. hint</td>
</tr>
<tr>
<td>Donald</td>
<td>3:44</td>
<td>2. prompt</td>
</tr>
<tr>
<td>Charlotte</td>
<td>7:45</td>
<td>3. hint</td>
</tr>
<tr>
<td>Potato Man</td>
<td>19:02</td>
<td>3. hint</td>
</tr>
<tr>
<td>Jim</td>
<td>5:43</td>
<td>1. clarify</td>
</tr>
</tbody>
</table>

Table 2: Think Aloud Performance on Shift Tasks

<table>
<thead>
<tr>
<th>Alias</th>
<th>The First Shift Task (see Figure 1)</th>
<th>The Second Shift Task (see Figure 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (minutes:seconds)</td>
<td>Help Given</td>
</tr>
<tr>
<td>John</td>
<td>2:03</td>
<td>0. none</td>
</tr>
<tr>
<td>Steve</td>
<td>1:48</td>
<td>1. clarify</td>
</tr>
<tr>
<td>Becki</td>
<td>1:05</td>
<td>0. none</td>
</tr>
<tr>
<td>Michael</td>
<td>1:24</td>
<td>1. clarify</td>
</tr>
<tr>
<td>Bobcat</td>
<td>14:36</td>
<td>3. hint</td>
</tr>
<tr>
<td>Lance</td>
<td>3:10</td>
<td>0. none</td>
</tr>
<tr>
<td>Johnstone</td>
<td>4:48</td>
<td>3. hint</td>
</tr>
<tr>
<td>Donald</td>
<td>3:44</td>
<td>2. prompt</td>
</tr>
<tr>
<td>Charlotte</td>
<td>7:45</td>
<td>3. hint</td>
</tr>
<tr>
<td>Potato Man</td>
<td>19:02</td>
<td>3. hint</td>
</tr>
<tr>
<td>Jim</td>
<td>5:43</td>
<td>1. clarify</td>
</tr>
</tbody>
</table>

Table 2: Think Aloud Performance on Shift Tasks

4. provide Providing a partial or complete solution if progress seems unlikely; or the subject has abandoned the task.

4 Dissection of Think Alouds
In this section, we dissect the think aloud sessions of Charlotte, Jim and Steve. Because of space limitations, we are unable to include the entire transcript for these students, and we have therefore chosen a selection of short excerpts which are representative of their attempts. Charlotte and Jim are typical of all four students who could solve the first task, but struggled with the second. Our presentation of each excerpt is broken into three subsections (summary, data, and analysis), following the format used by Lewis (2012).

4.1 Charlotte
Charlotte was in her third week of learning to program when she performed the following think aloud. This was her second think aloud session, and she was comfortable with the protocol of articulating her thoughts as she solved programming tasks. Charlotte possesses excellent language skills.

Charlotte took 7 minutes 45 seconds to solve the first task, with hints, and then spent 10 minutes on the second task before giving up. At the end of the think aloud, she was shown the solution; hence the “4. provide” for the level of help given.

4.2 Charlotte – The First Shift Task
4.2.1 Excerpt 1
Summary
Charlotte began by reading the problem. She initially expressed a lack of familiarity with the nature of the task. However, it was quickly established that she thought she was required to provide code to move the boxes. (In retrospect, not as bizarre an interpretation as we first thought, given the GUIs that students are now accustomed to experiencing.) The interviewer clarified that the task was to write code to shift the values in the variables according to the arrows in the diagram. To establish that Charlotte did then understand the task, the
interviewer asked Charlotte to choose some initial values for the variables and then determine the final values in the variables after her code had executed.

**Data**

Charlotte: So, may I ask is it similar to last week?

Interviewer: Yes, but instead of swapping two variables it’s ...

Charlotte: … swapping 4. And I want them all to move to the left. So I’m moving the values not the variables. Ok good to know - makes more sense.

**Analysis**

In this excerpt, Charlotte made a connection between shifting and swapping values: where each requires “movement” of values between variables using assignment. Although she used the word “swap” which is a reciprocated exchange of values between two variables, she showed an understanding of the shifts required.

### 4.2.2 Excerpt 2

**Summary**

Charlotte made a first attempt to solve this task and although each assignment statement in itself was correct (apart from using a variable t instead of temp) the sequence of her assignment statements was not correct. She then traced the code using the values she had chosen for each of the variables: 2,4,6,8 and 10 for a,b,c,d and t. When she incorrectly concluded that the code worked as required, she was challenged, and then decided to re-read the question.

**Data**

<Charlotte wrote the code below>

```plaintext
a = b
b = c
c = d
d = t
t = a
```

Charlotte: So it almost worked… Oh no! I think it did work the way I wanted it to. So it says the temp becomes 2. Yeah I think that worked.

Interviewer: Where does the value 2 end up?

Charlotte: <quoting the problem description> “…with the left most value ending in the right most variable”. Ah! It was cute while it lasted!

**Analysis**

Charlotte realised the importance of sequence and figured out that a’s value must be saved first, so that that value can be assigned to d after d’s value has been reassigned. Charlotte made the leap from individually correct assignment statements to correctly sequenced lines of code in order to achieve the required effect. She was however heavily reliant on tracing the sequence with specific values to convince herself of the code’s correctness, a manifestation of the preoperational stage of development.

### 4.3 Charlotte – The Second Shift Task

The second shift task was attempted by Charlotte in the same think aloud session where she completed the first.

### 4.3.1 Excerpt 4

**Summary**

Charlotte made a connection between this task and the previous task, but then had some doubt about their similarity when she read the supplied line of code. She established a set of initial values for each of the variables, and the expected final values for each.

**Data**

<As Charlotte uttered what follows, she wrote the initial and expected values in the boxes of the supplied diagram.>

<table>
<thead>
<tr>
<th>Variables:</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial:</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Expected:</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

Charlotte: So it’s the same as the first one. And then here that temp equals y, now I'm really sceptical. Um, I don't think it actually is, so we'll find out. 2,4,6,8 <values for variables w,x,y, and z respectively> and we want to move everything to the left and the left most one ends up in the right most variable.
**Analysis**

Charlotte manifests preoperational behaviour by setting up specific variable values with which she intends to reason about her code. Another preoperational behaviour is her focus on the superficial aspect of the task, that is, the initial assignment to the `temp` variable.

### 4.3.2 Excerpt 5

**Summary**

Charlotte paused to question the reason for the supplied line of code, but after not being able to come up with an answer, started to implement a solution. With the first assignment of `y` to `temp`, she articulated its new value, 6. When she had finished writing the remaining assignment statements (shown below), she was not confident that her answer was correct, and expressed frustration. To the left of each of her lines of code, she wrote the value being assigned to the variable on the left of the assignment. When the values didn’t match those expected, she realised her code must be incorrect.

**Data**

Charlotte: But you have to start with the `temp` as `y`. Why? Interesting question. … Fine. If you insist, `temp` is `y`, so `temp` becomes 6. … Where do I want it to go? Hmm. … Brain - wake up! … So … `x` to be `y` … Does that make sense? Ok for now it does. `w` to be `x` … `z` to be `w`. No we don’t. Nnnnn, yes we do. … Aaargh!

6. `temp = y;`
6. `x = y;`
4. `w = x;`
4. `z = w;`

*z* becomes 4 which we do not want! Think I’m breaking the thing I realised before.

**Analysis**

Although incomplete, most of Charlotte’s assignments were independently correct. However, the sequence of these assignments was not correct. She did not relate this second task to the approach she had successfully developed to solve the first task, but instead constructed assignment statements according to the diagram, in which she regarded each line of code as a separate assignment. Charlotte was unable to make an accurate determination of the code’s correctness until she traced it with specific values. Charlotte did not even trace her code accurately (in the third line she failed to take into account the updated value of `x`), and it was evident through utterances of contradiction (“No we don’t. Nnnnn, yes we do.”) and frustration (“Aaargh”) that she was cognitively overloaded. Because Charlotte said “Think I’m breaking the thing I realised before”, we hypothesise she had some hazy notion of the `invariant amid transformations` in this exercise, that is, that saving a variable’s value to a temporary location makes it possible to overwrite that value in the original variable. This was the “thing” that her current solution was "breaking".

### 4.3.3 Excerpt 6

**Summary**

Charlotte made her final attempt before running out of time. On this occasion, she started reassigning from the far right of the line of variables in the diagram and again recorded the value being assigned at each statement.

**Data**

Charlotte: `z` equals `w`, which basically becomes 2. `y` … becomes `x` so that’s 4. `<Expletive>` Sorry, `x` equals `y`, that becomes 6.

6. `temp = y;`
2. `z = w;`

6. `x = y;`

Um. Start over. `z` becomes `w`, that’s good because that’s 2. `x` becomes `y` which becomes 6 so that’s good. … Too confused … We have to back off here a little bit.

6. `temp = y;`
2. `z = w;`

6. `x = y;`

So we want `w` to equal `x` … which basically becomes 4. I haven’t removed `x`, the value of `x` yet. … I think that’s where things were trying to click in because then `x` becomes `y` … and that becomes 6. `y` becomes `z` which becomes 8. … Well … wait - what’s wrong with that? Why doesn’t that work?

6. `temp = y;`
2. `z = w;`
4. `w = x;`

6. `x = y;`
8. `y = z;`

Ok and `z` because we said `z` is `w` up here, so why is that a problem? … because that’s the problem! Grrrr! Ok, I think I have to go `<to another appointment>` …

**Analysis**

Charlotte’s piecemeal approach to solving this task was not effective. She was focused on individual assignment statements and lost sight of the bigger picture (shifting all of the values without losing any of them). She was unable to work with the cyclic series of variables starting from an intermediate point. For all the reasons given with these excerpts, Charlotte is clearly at the preoperational stage of development.

### 4.4 Jim

It was the third week of semester when Jim performed the following think aloud on the first task. Furthermore, in an earlier semester, Jim had successfully completed a course that included about 6 weeks of programming in Python. In his think aloud sessions, Jim demonstrated adequate language and communication skills. Jim had completed one think aloud session with us prior to completing the first shift task which is described below.
4.5 Jim – The First Shift Task

4.5.1 Excerpt 7

**Summary**
Jim read the question text and then proceeded to select values for each of the five variables.

**Data**
Jim: So we can say that a is 1, b is 2, c is 3, d is 4. And following what this diagram says, we also have a fifth variable which we will call e, though in the diagram it's called temp. That will be the value of 5. Though it doesn't matter.

**Analysis**
The diagram stipulated that the temporary variable was called temp. It is odd that he chose to rename it e. When later queried, he said he was opting for consistency: the other variables had one letter identifiers, so he chose a one letter identifier for the temporary variable. Also odd was his subsequent use of capital letters for the other variable names, instead of the lower case used in the diagram. In any event, as will be shown below, his unusual choice of variable names had no effect on achieving the desired outcome on this first task.

Jim’s reliance on specific values when reasoning about and writing code is characteristic of preoperational behaviour.

4.5.2 Excerpt 8

**Summary**
Jim articulated a logical sequence of assignment statements to complete the task, but was then not confident about his solution.

**Data**
Jim: So we want to move a first. So we want e to take the value of a. Um... then we can say ... that a can take the value of b. Um, c, uh b can take the value of c, c can take the value of d. And d can take the temp value. Jim had written the following:

\[
\begin{align*}
   e &= A \\
   A &= B \\
   B &= C \\
   C &= D \\
   D &= 5
\end{align*}
\]

...whoops. Going the wrong way around.

**Interviewer:** Have you?
Jim: Oh no I haven’t. So we want to go one more time around.

**Interviewer:** Do you?
Jim: To be ... well, we want A to be stored over here sindicating Đ>

**Interviewer:** What’s in D at the moment?
Jim: Um, in D at the moment is a 5.

**Interviewer:** Why did you hard-code ... the number 5?
Jim: Um, I just assigned it a value.

4.5.3 Excerpt 9

**Summary**
Jim was then asked to trace his code using the values he had already chosen. As he recounted each assignment statement’s effect with specific values, it was only then that he articulated the temporary storage and subsequent reassignment of A which convinced him that the code was indeed correct.

**Data**
Jim: So, e equals A so e will equal 1. A equals B so A will equal 2. Um B equals C, so B will equal 3. Um C equals D so C will equal 4 and D equals e so D will equal ... 1. Because e is equal to 1, that we'd gotten first at the top. ... Ok. So it's not 5, it’s 1. I see. So we have 1 in here <e> so that means we're going to have a 1 in here <D> now.

**Analysis**
Once Jim traced his code with specific values, he confirmed that his code was correct. Like most preoperational novices, Jim was not able to clearly reason in an abstract way about his code. He needed to trace the code with specific values in order to feel confident about its correctness.

4.6 Jim – The Second Shift Task

The second shift task was completed by Jim two weeks after he had done the first task. He took an enormous amount of time (more than 21 minutes) and several attempts to complete it. The following excerpts are only a small sample of Jim’s articulations for this task, but are representative of the difficulties he had.
4.6.1 Excerpt 10

Summary

After reading the question, Jim immediately recognised this task as familiar. He expressed scepticism about the given initial assignment statement. He then allocated values to each of the variables, including temp (both in the diagram and in the given line of code) and then worked his way through the diagram, writing an assignment statement to match each shifting value. He then attempted to formulate the correct sequence of those assignment statements.

Data

Jim: temp is assigned y. … This seems slightly unnecessary…

Ok um. So temp’s got the value of y … So … where are we… we’ve got … let’s say w equals 1, x equals 2, y equals 3, z equals 4. <He wrote the following set of initial values.>

\[
\begin{align*}
  w & = 1 \\
  x & = 2 \\
  y & = 3 \\
  z & = 4
\end{align*}
\]

So we want to move… we’ve got 1,2,3,4 … 3. No it’s easy, we get rid of that y value because we’ve got two 3’s. That means. So … um we can just say … Ok … so we want. … start <with> the y. … um … so we want … so we want … 1 … we want over here so we don’t want z to equal, z equals 1 then the 4’s going to disappear. If w equals x, the 2 is going to disappear. … If x, x equals y, the 3’s still going to … stay, so we can say… no the 2’s going to disappear so we can say y equals z. … So y equals z. <He wrote the following single line.>

\[
y = z
\]

So y equals z, so y will equal 4 now. So we’ve got 4 here … We can say… just wait. So still the left most variables … why would we want to do that, why wouldn’t we just say y equals … We need 3 so y equals… w. Going to move them all now. Um. What are we doing with this? I like to confuse myself a little bit. … And then we can have the 3 here, so it <z> is going to be … um 4 <recorded z as now having the value 3>. …

Yep. Ok. … Um … So we want x…. we want the z to equal w, we want w to equal z … z. We want x to equal y, and we want y to equal z. <He had written the following statements, separate from the previous single line of code.>

\[
\begin{align*}
  z & = w \\
  w & = z \\
  x & = y \\
  z & = z
\end{align*}
\]

So we’ve got y is equal to 4. So z is 3. So we want z to equal … 1, want w to equal 2, we want x to equal 3, we want z … z to equal w. <He revised the statements as follows>

\[
\begin{align*}
  z & = w \\
  w & = z \\
  x & = y \\
  z & = z
\end{align*}
\]

So … z is 4 so there we go <wrote 4 under the y of y = z>. That’s a bit … that's a bit better. So y to equal z. It's annoying because it's so simple, but not. [laugh]. Just messes with your mind!

Analysis

Jim determined that the reassignment of y should be the first step, only after testing the effect of first reassigning to z, then to w and finally to x.

Jim has so far made hard work of this task by recording four separate sets of data. First, he allocated integer values to each of the variables by writing what appeared to be assignment statements. Second, he wrote the beginning of an ordered sequence using those assignment statements. Third, he wrote an assignment statement for each “shift”, starting from the right hand side of the diagram. In addition, Jim kept current trace values recorded under several variable names in the code. Jim is dependent on reasoning with specific values in variables. With his trace notation interspersed in the code it was very difficult for him to follow on paper what he had written, let alone keep track of what he had left to do. When speaking, he repeatedly intermingled variables and values when referring to what needed to be assigned where. He made several contradictions by saying one thing and writing another. He showed some confusion about assignment direction, repeatedly changed his mind and made tracing errors throughout.

Jim was clearly cognitively overloaded, unable to manipulate the abstraction of the diagram in such a way that it represented a solution that started with the reassignment of y, and unable to design an effective trace of his code. These are all indicative behaviours of someone at the preoperational stage of development. Indeed, his haphazard approach to tracing is a characteristic of the sensorimotor stage. Although he did articulate an abstraction beyond the code itself, the need to “get rid of that y value because we’ve got two 3’s”, he did not continue to apply that principal to the remaining variables, as he had successfully done in the first task. Not applying an abstraction consistently and completely is characteristic of a preoperational novice.

4.6.2 Excerpt 11

Summary

At this stage, Jim had established expected final values for each of the variables, using the initial values he had chosen. After having painfully determined by trial and error what the first assignment should be, he struggled to establish a workable sequence of the remaining assignment statements.

Data

Jim: We want … x to equal the…3 so it currently holds the third value in temp. So we can say x equals temp. … So x has now got the third value. … temp is still empty so we can say… so we’ve got x and y sorted. Just need w. What do
we want \( w \) to equal? Whoops! <he exclaims while crossing out the third row below>. That shouldn’t be there because it gets rid of my 2 value.

\[
temp = y \\
y = z \\
x = temp \\
\]

So we need to store \( w \) in the \( temp \). \( temp \)’s got the value of \( w \) so now we can ... that \( w \) value. ... So that \( w \) value we want to equal 2... so we want \( w \) to take the value of \( x \). So the \( w \) value’s been wiped ... being stored in \( temp \), so the \( w \) value is given the value of 2 that should still be 2

\[
temp = y \\
y = z \\
temp = w \\
w = x \\
\]

[sigh] ... I think I just lost my ... lost my 3 then. Yeah, I’ve lost my 3 [sigh] Ah, it’s frustrating!

### Analysis

Jim correctly dealt with the reassignment to \( y \) after which he focused attention on the start of the series rather than continuing from that intermediate point. He struggled to implement the logic that he used successfully two weeks earlier on the first shift task.

In the first line of this excerpt, Jim refers to the "third" value, so we suspect that he saw the ordering of the variables in the diagram as significant. After dealing with the reassignment of \( y \) as required, he found it necessary to continue at the start of the diagram. This may explain his comment in Excerpt 10 that he found the forced assignment of \( y \) to \( temp \) as "slightly unnecessary". As a preoperational novice, he was unable to effectively apply the invariant of saving a variable’s value for subsequent reassignment. He had completed the first task successfully, but was unable to mentally manipulate the new diagram in such a way that it replicated the first, that is with \( y \) at the beginning of the reassignment sequence, rather than in the middle.

#### 4.6.3 Excerpt 12

### Summary

Jim made several other failed attempts at this task, experimenting with different values stored in \( temp \), but each time articulating a trace of the real values he had chosen. At a point where he was clearly frustrated, the interviewer suggested that he stop concurrently tracking the variables’ values while developing the code, thus eliminating what seemed to be a distraction.

### Data

**Jim:**

This is starting to frustrate me a little bit. [laugh] I’m not going to lie. Seems so much more um... I don’t know ... difficult. When you’re not doing it on the computer. What I’m saying is that ... like... if you don’t have the numbers there... you can ... I think

### Interviewer:

when you did this last week you ... stored one of the values away to start with. Why?

**Jim:** ...Um, well I don’t remember [laugh]

**Interviewer:** You don’t remember why?

**Jim:** Um, just so it didn’t get cleared. Ah, I see! ...Same as last week. I see ... But I’m just ... See what I’m trying to do, I’m trying to rearrange the numbers because I’m saying if its 1,2,3,4 .... and we’ve got the 3 in here <i.e. in \( temp \)> ...

**Interviewer:** So WHY do you have a 3 in there?

**Jim:** Because the \( y \) is equal to \( temp \). So, if I call \(<y>\) 3, then \(<temp>\)’s going to be 3

**Interviewer:** So then what’s your first step?

**Jim:** So the first step ... I can move the \( z \) to \(<y>\) ... And then I can move \(<x>\) to \(<w>\)...

**Interviewer:** ... when you did this last week, how many \( temp \) variables did you use?

**Jim:** One

**Interviewer:** So why should this be any different?

**Jim:** I don’t know. ... These \(<tasks>\) ... they’re like a lot easier than the programming that I’m doing, but they’re a lot harder at the same time. It’s just different - it’s weird. [laugh] It’s not nice. It confuses me.

### Analysis

Jim continued to have trouble with this task which forced him to start from an intermediate point, that is, the required initialisation of \( temp \). In the first task he appeared to have demonstrated an understanding of the process required to shift the values in four variables as well as the programming skills to implement it. However, without prompting by the interviewer, he had an enormous amount of difficulty transferring that (possible) understanding of a very similar task. His level of ability in terms of abstract reasoning was clearly preoperational.

#### 4.7 Steve

Steve’s think aloud sessions were indicative of concrete operational reasoning. Steve was in his first semester of learning to program. He completed his first think aloud session in week 3 of semester.

### 4.7.1 Excerpt 13

### Summary

After needing initial clarification of the diagram, Steve completed the first task in a matter of seconds.

### Data

**Steve:** So a will become d and d will become a

**Interviewer:** Ah, the value in a will go into d - like this diagram shows, the value of a eventually goes to d.

**Steve:** and d eventually goes to a.

**Interviewer:** ...c goes into b, b goes into a...
Steve: Ah, so shuffle it along.

Interviewer: Yeah. Move everything up to the left

Steve: Ok so... temp equals a. a equals b. b equals c. c equals d. d equals temp.

Analysis
Steve’s initial interpretation of the first task was that the values in variables a and c were to be swapped, with the top arrows in the diagram indicating the passing of c’s value through a and b, and finally ending up in a. His understanding was quickly corrected, confirmed by his articulation of the task as a ‘shuffle’ and then immediately writing a correct solution.

4.7.2 Excerpt 14
Summary
Steve then attempted the second task, and completed it without hesitation:

Data
Steve: Ok... temp equals y so we’ve stored the y value. So then we can replace it with the z value. Yes, y equals z. Then you replace the z value with w, w value with x... And then. Ah yeah, then x value with the temp

Analysis
Steve had clearly identified the invariant: “temp equals y so we’ve stored the y value”. He applied the same process of storing a value before overwriting the variable with what was to replace it, for the remainder of the variables. With concrete operational skills, Steve had no problem applying the skills he used in the first task to the slightly different second task.

5 Discussion
During these think aloud sessions, we noticed variation in the way that some students articulated assignment statements. For example, with respect to the following assignment statement:

```
a = b
```

some students articulated the statement from left to right, thus:

“a is assigned the value of b”

others read from right to left, that is:

“the value of b is assigned to a”

while others articulated assignments both ways: sometimes left to right and sometimes right to left. We conjecture that such variation in articulation is indicative of novices at a neo-Piagetian stage lower than concrete operational.

During the think aloud sessions, it also became apparent that some students struggled to process the diagrammatic depiction of the problem. One possible problem was the direction of value "shifts", as the majority of the values passed between variables right to left, but the value originally in the leftmost variable moved left to right. Some of the students even expressed confusion over the meaning of the arrows. Apparently it was not immediately clear (as it was to us, and probably to any experienced programmer) that the arrows indicate the direction of movement of the values.

The think aloud students who struggled with the second shift problem tended to look at a small part of the diagram and implement it. Next they would return to the diagram and find another piece to implement, without much thought to the consequences of sequential execution. They had not developed an overall design for their solution, but instead focussed on the functionality for each independent piece of the problem, in the hope that they would somehow all fit together in the end. Being distracted from the most salient aspects of the problem by individual elements is characteristic of preoperational reasoning.

Even some students who completed the second task quickly expressed some awkwardness about it. Lance said "That felt weird. I didn’t really like having to start there. Don’t know why." Becki said that the second task was “very sneaky” and it had ruined her plan to start from the end as she had in the first task. She also said that it would not have made a fundamental difference had the diagram depicted the variables in a circle as the variable names were ordered and she tended to work on the variables in lexicographic order. However, despite some initial and brief confusion, these students were able to complete the task. Students like Lance, Becki and Steve thus manifested concrete operational skills.

6 Conclusion
In this paper, we have presented data from a think aloud study which demonstrates that some novice programmers manifest behaviours characteristic of the preoperational stage in neo-Piagetian theory. One such behaviour is that they tend to focus on parts of a programming task and lose sight of the task as a whole. Students who struggled with the second “shift” task tended to examine a portion of the diagram and implement it, then return to the diagram and find another portion to implement, and so on, without considering the overall sequence of execution.

Another characteristic of these preoperational novices is that they are prone to focus on superficial aspects of a specific task that are not salient to solving a general class of tasks. In neo-Piagetian terms, preoperational novices do not focus upon aspects of tasks that are "invariant amid transformations" (Flavell 1977). In the “shift” tasks, the invariant is the idea of duplicating a variable, so that the value in the original variable might then be overwritten, while the superficial aspect of the task is the initial assignment to the temp variable.

These two characteristics lead preoperational novices to adopt an approach that might be called programming by permutation. On very small tasks, that approach may indeed lead the novice to a correct solution, especially if they are completing that small task on a computer and thus receive feedback by running their code. However, novices who adopt that approach do not learn abstractions that they can then transfer to a very similar task.

The two “shift” tasks we gave our students are very simple programming tasks, the solution for which is near-identical in most imperative languages. The problems experienced by some of our novices are therefore not
caused by the particular programming language in which they write.

Piaget (1971b) described reasoning at the preoperational stage as that “…which consists simply in retracing … events just as they were perceived, instead of imagining an alteration …”. It is only at the concrete stage of development that novices develop the ability to work with cyclic series, to reason about abstract quantities that are conserved, and transfer a general approach to a slightly different task.

When students demonstrate difficulties with programming, it may not be a reflection of their innate ability to learn programming, but rather an indication of their current state of cognitive development. Struggling students may not have yet developed the mental schemas necessary to perform at the concrete operational level of reasoning required by certain programming tasks.

On the basis of our qualitative work, we cannot draw firm conclusions about the commonality of preoperational reasoning. However, given that four of our eleven think aloud volunteers manifested this difficulty, it is possible that preoperational reasoning may be common. Further quantitative work is warranted. If future studies confirm that this is a widespread issue among novice programmers, then it suggests that our teaching practices should change. The change would place the focus on identifying the current neo-Piagetian stage of a novice, and provide tuition appropriate to moving that novice to the next stage. Current pedagogical practice places little emphasis on the sensorimotor stage and completely ignores the preoperational stage. That is, current pedagogical practice assumes that the basic programming constructs are learnt easily, and then students immediately begin to reason about programs at the concrete operational stage.

7 Acknowledgments

The authors thank the students who volunteered for this study. Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office for Learning and Teaching or the Australian Government.

8 References


Teaching Mobile Apps for Windows Devices Using TouchDevelop

Simon
University of Newcastle
simon@newcastle.edu.au

David Cornforth
University of Newcastle
david.cornforth@newcastle.edu.au

Abstract
Even in a computer science degree, some students find it very hard to learn programming. In a less programming-oriented degree such as information technology the problem is amplified, and it is a real struggle to engage some students with the programming courses. A wealth of literature describes various approaches to teaching programming in the hope of addressing the perennial learning problems and encouraging the students to engage with the material. One approach to engagement is to teach the most current material, and one form of material that is highly current is programming apps for mobile devices. In this work, we report on one approach using a new programming language that was specifically designed for mobile app development. The approach was a success, with students becoming engaged. However, there are issues which we hope to address in the future.

Keywords: native mobile apps, programming education, TouchDevelop

1 Introduction
In the Bachelor of Information Technology degree at the University of Newcastle, Australia, the focus on learning programming skills is somewhat less than would be expected in a computer science degree. While all students must complete the first programming course, subsequent programming courses either form parts of particular majors or are entirely elective. In this context, it was decided that the second programming course would teach the development of apps for mobile devices, in the hope of attracting students who might otherwise take no programming courses beyond the first.

Jackson et al (2013) discuss reasons for teaching mobile app development, including relevance to business, the ease of programming games, the appeal to students who work all day with mobile devices, and a response to the challenge of retention. Another compelling reason is that the teaching of programming should reflect the current trend in computing platforms away from the desktop and towards mobile devices. As Tillman et al (2011) note, more touchscreen devices were sold in 2011 than laptop and desktop computers combined. It is important that graduates be made aware of this, and of the challenges it poses for the IT professional.

Mobile apps can be developed as either web apps, which are accessed by way of a web browser on the mobile device, or native apps, which are compiled expressly for the device in question. Web apps have the advantage that they can be accessed on any web-enabled platform, while native apps must at best be recompiled for each target platform, and in many development environments must be largely rewritten to suit a particular platform. On the other hand, web apps may be generic apps that have limited access to the hardware of the phone, whereas native apps tend to offer far greater access to the hardware and software specifics of the platform for which they are written. Whichever of these approaches is chosen, there is also the issue of testing the app on a mobile device and deploying it to the device. For this course we wanted to give students an awareness of both the web apps and the native apps approaches, including interaction with cloud services, as well as broad exposure to the device-specific features such as accelerometer, compass, contact lists, etc.

2 Android, iOS, or other?
When contemplating a native mobile apps programming course, the choice of platform is crucial for several reasons. Goadrich and Rogers (2011) discuss the range of development platforms and languages for native mobile apps, and conclude that there are two obvious options: Apple’s iOS to develop apps for various Apple devices (such as the iPhone), and Android to develop apps for Android devices. After conducting a detailed comparison of the two, they conclude that while iOS probably has more teaching and learning overheads than Android, either is suitable. However Skelton et al (2013) note the need for an iOS or Android app to go through a review process before being made available to other users. This would represent an unwelcome delay for many students.

It is probably reasonable to expect that many of the students will already have an iPhone and the rest will already have an Android phone. Each group will presumably therefore feel disenfranchised if the course teaches programming for the device that they don’t have.

However, the teaching and learning overhead implicit in the decision is far more important than the consideration of which students have which device. Notwithstanding the various IDEs available with either platform, programming an Android device tends to entail programming in Java, and programming an iOS device tends to entail programming in Objective C (Skelton et al 2013). If the course objectives include teaching the students a new programming language, the choice can be made in that light. But if the principal objective is to
expose students to the concepts of mobile apps programming, there could be virtue in choosing the language that has least additional overhead for the students.

Various authors (Roy 2012, Dabney, Dean, & Rogers 2013, Honig 2013) observe that Android is the obvious choice if the goal is to reduce the additional cost of learning a new language. However, they do this in the assumption (sometimes not explicit) that the students will already have learnt to program in Java; and while this assumption holds good for many computer science degrees, it is not universally valid.

In the Bachelor of Information Technology degree at the University of Newcastle, Australia, the first programming course uses the Python variation of the Media Computation approach of Guzdial and Ericson (2013). Some of the students will additionally take an introductory software engineering course that uses Java, but this is by no means given, and indeed the software engineering course is not offered on all campuses of the university. Therefore it cannot be assumed that the students enrolling in the mobile apps course will have programmed in Java; and experience shows that most of them have not.

Both Objective C and Java are so different from introductory-level Python that a course using either will need to devote considerable time to teaching the language, leaving less time to teach the mobile apps concepts that are intended to be the focus of the course.

There are many options, some of which can be deployed on several platforms without the need to rewrite code. Some of these environments are conveniently summarised by Smith (2012). It was through this summary that we became aware of the new TouchDevelop language (Horspool & Tillman 2013), developed by Microsoft for Windows mobile devices. This language seemed significantly simpler than either Java or Objective C, suggesting that it could be taught alongside the mobile apps concepts rather than as a prelude to them. Moreover, it permitted a neutral approach to the students’ own devices, as Windows phones are by no means common among the students at our university.

After some time spent exploring the TouchDevelop language, we therefore decided to proceed with the course using this language. Further reasons for this choice were:

- Requiring implicit rather than explicit knowledge of OO concepts
- Powerful built-in features
- Simple deployment to device
- Simple access to phone sensors
- Common development environment on web and phone
- Simple sharing of developed code
- Online community

Discussing best practices in mobile app development, Mahmoud (2011) stresses the importance of providing students with mobile devices, and mentions academic programs “that provide devices free of charge to academic institutions interested in integrating mobile devices and mobile application development into their curriculum.”

Our course proved no exception in this regard: Microsoft contributed a number of phones that were close to being supplanted by newer models, and we ended up with enough for each student to borrow one for the duration of the course.

3 The TouchDevelop Language

TouchDevelop has been developed to create mobile apps that can then be deployed on a mobile device. However, unlike other mobile app development languages, a central feature of its design is the ability to develop code using the mobile device itself as a platform. A TouchDevelop program is written not by typing text but by tapping tiles, the usual means of interacting with software on a mobile device; and this can be done just as easily on a smart phone as on a desktop or laptop computer.

Notwithstanding the design intention, writing code on a mobile phone can be fiddly. Fortunately, it is also possible to develop code on a TouchDevelop web application, and the teachers and the students all agreed that this was by far the preferable approach.

TouchDevelop is cloud-based, and programs (scripts) and other data are stored on the cloud. Access is by way of a suitable account (Microsoft, Google, Facebook, or Yahoo), and synchronisation is automatic: a script developed on the web application is available more or less immediately on the phone by way of the same account. Developing a script on the web application and testing it on the phone is a reasonably seamless sequence.

Goadrich and Rogers (2011) illustrate Android and iDevice development with a simple program that displays random permutations of the letters of “Hello world”.

Their iOS project consists of four files. They write of one of these files: “The syntax may appear daunting at first blush, but it takes just a few minutes in the classroom to explain to students familiar with OO-ideation” (Goadrich & Rogers 2011). They go on to emphasise that the complex-sounding process of creating the project is actually quite simple once the programmer is accustomed to it.

Their Android code, also in multiple files, includes at least 45 lines of xml and 30 lines of Java.

By way of contrast we illustrate TouchDevelop code with a script that does the same thing – though for good measure, the permutation is activated not just by a button press, as in the examples of Goadrich and Rogers, but also by shaking the phone. As shown in Figure 1, there are 17 lines of code (one line is wrapped to fit in the figure), in a single file. Even then, the code is more strung out than it needs to be: it would be easy to write a shorter version. This example alone should make it clear that TouchDevelop programming is simple in comparison with iOS and Android.

Our students at the University of Newcastle cannot be assumed to be familiar with OO concepts or with the languages used in the sample projects of Goadrich and Rogers. Furthermore, in their first programming course, every program that they encounter is written in a single file. For these reasons, among others, TouchDevelop would appear to be highly suitable as a next step for these students.
3.2 Writing TouchDevelop code

As indicated earlier, TouchDevelop code is written not by typing but by tapping tiles on the screen – or, when using a computer, by clicking them with the mouse. Even when a line of code is made up entirely of characters that can be found on the keyboard, an attempt to type that line will probably fail: the application does not always recognise, say, that the three letters ‘v’, ‘a’, and ‘r’ are intended to be the same as the ‘var’ token. Keyboard editing of an existing line of code is particularly troublesome.

Figure 1: Complete TouchDevelop script for permutations of “Hello, World!”

```
action main()
    wall → add button("questionmark", "Permute")

private action permute()
    var str := "Hello, world!"
    var newstr := ""
    for 0 ≤ i < str → count do
        var length := str → count
        var pos := math → random(length)
        newstr := newstr || str → at(pos)
        str := str → substring(0, pos) ∥
            str → substring(pos + 1, length - pos - 1)
    newstr → post to wall

event shake()
    ▷ permute

event tap wall Page Button (item : Page Button)
do
    ▷ permute
```

4 Positive Aspects

We found a number of positive aspects to the use of TouchDevelop as a development environment for the second programming course.

4.1 Using the phone to write code

It is potentially advantageous to be able to program on the phone itself, although the extra fiddliness of this approach means that the option of programming via the web application is generally preferred.

4.2 Web development environment

As a web application, the development environment requires no installation, just a reasonably current browser. The cloud storage for scripts means that one’s scripts are available from any device at any time.
4.3 Language design
By and large, we found the programming language to be well designed. In particular, the language designers were prepared to throw away some (though not all) of the ill-conceived features of the C-based languages, such as the use of the equality symbol from mathematics to mean something very different from equality.

4.4 Ease of code sharing
On creation, every app is given a unique code of four or more letters. To publish the app entails just a few simple steps: there is no code review or similar barrier to publication. Once it is published, any user can search for and install the app with a given code. When our students submit apps, they tell us the corresponding codes. We can then use those codes to install and assess the apps, and can pass the codes to other students so that they can examine the apps and provide constructive comments on them.

4.5 Online developer community
TouchDevelop is more than just a programming language: it is an online developer community. Students are exposed to the collaborative nature of the programming task, and are encouraged to examine other scripts and learn from them. We played to this feature by requiring all students to comment on ten draft scripts produced by other students, and to provide considered responses to the comments on their own scripts. Many students at first considered this an unwelcome imposition; but by the time the assignment was complete most of them appear to have risen to the challenges, both providing positive feedback on other students’ scripts and responding well to the feedback on their own.

4.6 Environment for further development
For programmers who feel the limitations of the language, it is possible to export a TouchDevelop script written for Windows 8 in a format that can be loaded as a project into Visual Studio. In the future we plan to explore the option of continuing development of a script in that environment, either through this means or by using TouchDevelop as a vehicle for rapid prototyping.

4.7 Novice-friendliness
The TouchDevelop environment is designed for and well suited to novices. The error reporting is excellent, and the immediate yet unobtrusive help makes it difficult for students to generate erroneous code, and extremely easy for them to diagnose and correct errors when they do arise.

This feature makes TouchDevelop ideal for the purpose for which we are using it: teaching the students to write apps for a mobile device, while at the same time teaching them a new language almost by stealth.

4.8 Support
As the students and teaching staff confronted a number of issues with the TouchDevelop language and environment, we were fortunate to have contact with people at Microsoft who responded very quickly to our requests for help.
5 Issues

Offsetting the positive aspects of the language and the development environment, we certainly had some concerns that affected both the academic staff and the students. However, the language and the documentation are in constant change. Having taught our course in semester 1 of 2013, we expect that some of the problems we encountered will have been remedied by the time we next teach it in semester 1 of 2014.

5.1 Code input

When using the web application from a normal computer, the inability to type most code on the keyboard can be frustrating. This is not such a problem when entering symbols that are not found on the keyboard, but is disconcerting, to say the least, when entering what appears to be straight keyboard text. For example, the expected way of entering a string is to select the "abc" token on the far left of the set of touchable tokens. This opens a new line below the line that is currently being written. The string is entered in that line, without quotation marks, and entry is terminated by touching or clicking back in the line being written. The string then appears in the correct place in the command, complete with quotation marks.

A keyboard-savvy programmer entering the same line of code might well type a quotation mark. This opens the same ‘string-entry’ line below the line being edited, but the programmer is possibly confused that the quotation mark is not shown. She proceeds to type the string, ending with a closing quotation mark; but anything typed in the string entry line is taken to be part of the string, so when eventually the programmer clicks back in the statement, the typed quotation mark has become part of the string, and gives rise to a syntax error – or, more recently, to a string that includes a spurious quotation mark at the end.

It doesn’t take long for most people to learn to click or touch even when typing looks possible; but die-hard keyboard users continue to be frustrated by problems such as this.

5.2 Language in development

The language is still in development, and is being changed quite frequently. For example, during the semester of our course, the language appears to have dispensed with a particular type of layout box and the possibility of creating a tile to launch the app. We were able to find an undocumented workaround for some deprecated code, but this is presumably a temporary fix.

The reference book provided on the TouchDevelop site is almost of necessity perpetually out of date. It includes deprecated code that cannot be entered, describes features that are no longer in the language, and fails to mention new features that are. It is very clearly a reference book rather than a textbook. Some of the examples in the book are not very well thought out, or not very well programmed, or both. On the other hand, it is (at the time of writing) free, which is great advantage for the students.

5.3 Inconsistencies between web and phone

While programming on the web application is generally preferable to programming on the phone’s TouchDevelop app, there are significant differences between the two. Some of these are quite subtle. For example, the code in figure 1 includes the expression str->substring(pos+1, length – pos – 1). If the –1 is omitted, the expression defines the substring from index pos+1 to one beyond the end of the original string. The web application deals with this comfortably, returning the substring from index pos+1 to the end of the string. But when the same script is run on the phone, it correctly generates an index-out-of-bounds runtime error. Some of our students chose not to borrow a phone (one student explaining that he already has too many phones), and were therefore unaware that the code they were handing in might not run on the mobile device even though it runs on the web application.

A related problem is code that runs on the web application but simply does not exist on the phone app, either because it is a new feature that has not yet been implemented on the phone or because it is an old feature that has not yet been removed from the web application. Code in this category does not simply fail to run on the phones: it fails to appear on the phone’s list of scripts available for download from the cloud. A programmer might write four scripts on the computer. Within minutes, three of them can be found on the phone; but the fourth simply never appears. We eventually discovered how to find out why this was happening, and how to check whether a script had such problems. But because it was well into the semester when we found these things, some students never caught up with the discovery, and ended up submitting code that would not port to the phones.

Any script developed by editing another script comes with a history, and can be traced right back to its first ancestor. Perhaps the worst aspect of the problem described above is that if a script has code that prevents it from porting to the phones, every descendant of that script inherits the condition – even if the offending code is removed. Thus a script with no phone-incompatible code will fail to port to the phone because one of its ancestor scripts had some phone-incompatible code. All we could do for students in this position was suggest that they start a completely new script and enter all of the code again.

5.4 Cloud availability

There were times when part of the TouchDevelop cloud itself appeared to go offline for several days at a time. At such times it was still possible to write code on the web application, but it was not possible to log in from a mobile device or to access other people’s work. This happened only two or three times during the semester, but, perhaps inevitably, one of those times was a weekend when students’ work was due to be submitted.

5.5 Graphics

One issue that gave us serious concern was the matter of graphics. Any graphics used in a TouchDevelop script must first be uploaded to the cloud; the script then downloads the graphics to the device each time it is opened. All graphics uploaded to the cloud in this manner
become accessible to all TouchDevelop developers. While each graphic can be accompanied by a comment explaining who created it and who uploaded it, this information is not mandatory, and the TouchDevelop cloud is therefore host to a growing number of images with varying degrees of attention paid to their intellectual property and to associated privacy issues. Suppose, for example, that a student wishes to write an app that cycles through a series of photographs of her family while playing an appropriate soundtrack: then those photographs must first be uploaded to the cloud and become public property. From the academics’ point of view, this has the potential to teach students an important lesson about privacy and intellectual property. However, especially under the pressure of deadlines, it also has the potential to encourage laxness with regard to these same questions.

5.6 Language design

Finally, while we have great respect for the language designers, we do wonder about some of the design decisions. For example, while the language has collections, it appears not to have arrays, a staple of programming for many decades. And while it has a for loop, the index of that loop must be integer, can start only at zero, and must count upward in ones. Of course it is possible to write the expression that converts from this constrained index to the desired sequence. But we would suggest that if one wants to count backward from 100 to 34 in steps of three, it is clearer to have an index going from 100 to 34 in steps of –3 than to have an index going from 0 to 22 in steps of 1 and to refer constantly to 100 minus three times that index.

It must be noted that the course was taught using TouchDevelop for Windows 7 devices. By the time this paper appears, that product will have been discontinued in favour of TouchDevelop for Windows 8 devices. The reference book has already been updated, and we anticipate that some of the issues discussed here will have been addressed in the new version of the language and platform.

6 What Did the Students Think?

The overall impression of this new course is that it was a success. It was clearly engaging, and most students appreciated the opportunity to program apps for a mobile device, although of course some would have preferred it to be for their chosen style of mobile device.

The main reason for choosing TouchDevelop over iOS or Android development environments was simplicity of the language, knowing that many of our students had programmed only in Python in the context of media computation. In this we clearly succeeded: all of the students who persisted with the course did well, and few of them had serious problems coming to grips with the language.

In a somewhat complicated transition arrangement, many of the students who took the first offering of this course had already completed two programming courses, typically in C#, and many of these students found TouchDevelop too simple. Pertinent comments from the standard end-of-course survey (possibly all from the same student) include:

- The skills taught in the course in no way apply to the real world due to the fact that TouchDevelop is a hobbyist language for kids and is in no way used in the real world.
- The course was far too easy and that was due to TouchDevelop being used, go back to industry standards like C# or Azure, a course like this in no way helps me with skills I need in the workplace.
- Teaches students to make small scale novelty apps for a platform that has no weight in the real world and the skills from TouchDevelop are not transferable as the skills are very basic in that they have been learnt in previous [first-year] courses.

On the other hand, some students appreciated the simplicity of the language:

- Good content, good pace and understanding of student engagement. Didn't overload us or cram too much work into any one lecture.
- Actually, learning about APIs was pretty nifty piece of information.
- Touchdevelop can be annoying at times, but it is a really easy language to develop prototype in.

Students definitely picked up on the fact that TouchDevelop is a work in progress:

- Touchdevelop is still in beta with copious amounts of bugs, NOT suitable for a professional university course. Even if it were at production level it then might be suitable.
- The platform for this course is under-developed, there are many problems which needed to be worked out, and would be inclined to postpone the use of the current platform until it is improved ... In future it could be a good platform to use, but as of yet, I would postpone the use of it.

Notwithstanding the clear negative comments, it does appear that the overall impression was positive:

- This subject is useful for student who interested in coding language. I realized that modern new TD language has similar concept with other coding language but it has wide as wide range of application so I interested in learning this course.
- This has actually been one of the courses I have enjoyed most at uni so far ... It doesn't feel like we are learning archaic concepts either, which makes it more motivating.

In order to address the comments of students, we are considering changes to the course for next year. The perception that the course was too simple will be largely addressed by completion of the current transition phase. In future years, almost all students entering the course will have only the minimal programming background of a single semester using Python. For those students with more advanced programming skills we plan to allow a more advanced option where students may choose to complete their project using Visual Studio instead of TouchDevelop. However they will still be required to use TouchDevelop as a prototyping tool. This will allows us to offer a challenging path, using JavaScript and HTML, for advanced students, while still meeting the needs of students struggling to master the basics of programming.
The course material will of course be more highly developed for the second offering, so this should address some of the other issues that were raised by the students.

7 Discussion and Conclusions

Responding to the need to rework the second programming course in our information technology degree, we decided that a course in programming mobile apps would engage the students and persuade more of them to continue their programming education. However, we faced a substantial problem: having learnt only one semester of programming, using Python in a media computation context, the bulk of our students had no familiarity with any of the C-based languages.

The newly designed and implemented TouchDevelop language appeared to offer a solution to this problem. It is a simple language with robust error checking, and its use could free us from having to devote large amounts of time to teaching a programming language as opposed to the concepts and techniques of programming apps for mobile devices. Nguyen et al (2012) have empirically shown the efficacy of code produced, and the positive impact that this has on productivity. They showed that students using TouchDevelop completed more tasks than those using Android, and that the likelihood of a task being completed was greater for those using TouchDevelop than for those using Android.

Athreya et al (2012) observe that TouchDevelop is an appropriate environment for producing small apps that have simple features, but that are reliable and easy to get working. This provides further evidence of the relevance of TouchDevelop for teaching programming. Athreya et al (2012) also comment on the wide variety of scripts that users typically produce when using TouchDevelop. This is of benefit when offering projects to students for assessment. We find ourselves in clear agreement with these observations.

While we do have issues with TouchDevelop, the overall impression of both staff and students is that this was a good approach to teaching mobile app programming. Furthermore, the existence of a clear developer community led us away from the punitive develop-code-in-isolation model to one that acknowledges the role played by peers when developing code.

It is clear to us from the first offering of the course that, as covered in the reference book, there is not enough material for a full-semester course. In the next offering we hope to extend the course to add interaction with cloud-based data and to cover further development of TouchDevelop scripts in Visual Studio.

In closing, we should address one question that might arise when reading the students’ feedback on the course: if TouchDevelop is so simple, why not use it for the first programming course rather than holding it back for the second? We have given this question serious consideration, and decided that this is not a change we are prepared to make. The students’ comments were made in the context of having completed one, two, or possibly more programming courses. We do not believe that complete novices would find the language so easy to pick up. Indeed, we believe that the tile-based nature of the programming and the link to mobile devices might distract novice students from the essence of programming, which is where we focus in the introductory course. And finally, we think it unlikely that we would ever have enough phones to be able to lend one to each of our first-year students. For these reasons, we confidently expect that this course in programming for mobile devices will remain our second programming course for at least the next few years.

8 Acknowledgements

The authors would like to thank the Microsoft TouchDevelop team for the provision of phones for use in the first offering of this course, and for the generally impressive level of support provided during the course.

9 References


A Web-based Firewall Simulator Tool for Information Security Education

Zouheir Trabelsi and Umniya Mustafa
College of Information Technology
UAE University
PO Box 17551, Al Ain, UAE
{trabelsi, umniya.M}@uaeu.ac.ae

Abstract
Teaching practical information security requires the use of techniques, security and network devices and software, simulator tools, testbed networks, and hands-on lab exercises to support the educational process. This paper presents an educational web-based firewall simulator tool to help students learn the intricacies of writing firewall filtering rules to filter and inspect network traffic. The design principle of the simulator tool is to be easy to use while teaching the students the details of writing basic and advanced filtering rules. The simulator tool offers a set of educational functions that are not commonly available in professional firewalls. The tool can be used in any network and security course by instructors and students in the classroom. The impact of offering the simulator tool on the students’ performance in terms of achieving the course outcomes is also discussed.

Keywords: Information security education, Firewall, Packet filtering, Filtering rules, Simulator tools.

1 Introduction
Security has become a major concern throughout the world. Increasing information security education and teaching knowledge of security techniques to students who major in computer science and information technology, and satisfying new expectations for information technology professionals has become a very urgent need (Conklin 2006 and Al-Shaer 2009).

Moreover, the need to use a practice and application oriented approach in information security education is paramount. A security education curriculum that does not give the students the opportunity to experiment in practice with security techniques cannot prepare them to be able to protect efficiently the confidentiality, integrity, and availability of computer systems and assets. In addition, teaching practical information security requires the use of techniques, security and network devices and software, simulator tools, testbed networks and hands-on lab exercises to support the educational process.

On the other hand, since firewalls are an import security topic, information security programs should include courses that cover firewall concepts and technologies. According to students’ feedback (Williams 2011), after traditional lectures on firewall concepts some of them have difficulty fully understanding the use and configuration of a firewall. For many students an interactive education tool can help them to understand the functions of firewalls by getting hands-on and configuring a firewall step by step.

Therefore, to enhance the student’s practical skills on firewalls, schools offering information security programs need to acquire professional firewalls in order to offer to the students hands-on lab exercises on firewalls. However, professional firewalls are designed mainly for professional uses, and are not adequate for the academia environment. That is, most of the offered functions in professional firewalls are not needed for information security education and do not allow students to better anatomize the firewall concepts and technologies taught in the lecture. For example, most professional firewalls do not allow the user to directly manipulate and freely set the values of the fields in a filtering rule. In addition, firewalls are usually expensive hardware devices; however academic institutions have limited budgets to setup security lab facilities.

In order to enhance information security education, firewall simulator tools can be a very effective solution, and can help students better understand the functions of a firewall and filtering rule implementation. Also, firewall simulator tools let students get hands-on experience and are ways of learning the practical aspects of firewalls. In addition, compared to professional firewalls, firewall simulator tools are an affordable solution for schools offering information security programs with limited budget for setting up security lab facilities.

This paper discusses the design and implementation of a web-based firewall simulator tool, called Edu-Firewall, that is dedicated for educational purpose. Edu-Firewall tool offers basic firewall functions as well as advanced educational security functions that are not available in current professional firewalls. Moreover, compared to professional firewalls, Edu-Firewall tool allows students to better anatomize the advanced concepts of writing effective and efficient firewall filtering rules for standard and nonstandard services. In contrast to many professional firewalls, the tool offers a graphical user-friendly interface that can be used to create advanced and low level filtering rules, and to freely set the values of the fields of filtering rules.

The paper is organized as follows: Sections 2 and 3 discuss related simulator tools, and limitations of professional firewalls, respectively. Section 4 discusses the design considerations of the proposed firewall simulator tool. Sections 5 and 6 describe the educational
functions and the implementation of the tool, respectively. Section 7 discusses the effect of offering the tool on the student performance, and the evaluation of the learning outcomes. Section 8 discusses the students’ satisfaction. Finally, Section 9 concludes the paper.

2 Related Simulator Tools

Firewalls are security devices or software tools used to apply an organization’s security policy by implementing filtering rules to filter the incoming and outgoing network traffic. A number of educational firewall simulation systems have been previously developed. The goal of some systems (Garrido 2009 and Ye 2006) is to provide the student with a statistically supported understanding of a firewall’s effectiveness. The configuration of the firewall in these systems is fixed by the simulation and not defined by the student. These systems give the student an understanding of network and firewall load, but provide very little training on the configuration of a firewall. Several systems (Wang 2010, Hu 2004, and Stewart 2009) make use of virtualization for security simulations. Virtualization is effective for providing each student virtual resources, including firewalls, that the student can deploy to secure a network.

CyberCIEGE (Irvine 2005) is an interactive game designed to teach computer and network security concepts. It provides the student with a wide range of security threats and possible solutions. Firewall configuration within CyberCIEGE is done at a high level without defining the details of what specific packets are to be filtered. Students are required to configure a simulated firewall using Cisco-like commands to prevent other students from attacking them.

The objective of the interactive firewall simulator presented in (Williams 2011) is to provide students with an interactive competitive system to help them better understand the concepts of firewall configuration and operation. However, this tool provides limited number of educational functions to optimize the filtering rule set, such as the verification of the consistency and efficiency of filtering rules. Also, the tool does not allow students to implement low level filtering rules, and better anatomize common advanced firewall concepts, such as stateful packet inspection, nonstandard service filtering, and packet content inspection (known also as Deep Packet Inspection (DPI)).

All the above related simulator tools have a common limitation regarding the covered firewall concepts. That is, they offer students basic firewall functions for firewall configuration and simple filtering rules implementation. However, students enrolled in an information security program require to anatomize basic as well as advanced firewall concepts and writing low level filtering rules.

3 Limitation of Professional Firewalls

This section describes a comparison study that we conducted to evaluate the available educational functions offered by four common professional firewalls (Cisco ASA 5520, Juniper Networks, Preventia and Jetico Personal Firewall). The main objectives of this study are to identify the ability of the firewall interface to allow its users to:

1. Implement low level and advanced filtering rules for packet filtering and content inspection.
2. Visualize the status of the network stateful sessions.
3. Identify and correct inconsistent and inefficient filtering rules.
4. Assign a wide range of value types to the filtering rules’ fields.

Table 1 shows the results of the study and the limitations of the evaluated professional firewalls. In fact, when they are used by students for hands-on lab exercises, the evaluated firewalls do not allow students to (1) freely set the values of the fields in the filtering rules, (2) write low level and advanced filtering rules and (3) verify the consistency and efficiency of filtering rules. Consequently, the evaluated firewalls do not allow students to better anatomize the advanced firewall concepts. This study confirms the claim that professional firewalls are designed mainly for professional uses, and are moderately adequate for the academia environment.

In addition, small multi-port wireless routers are for general purpose use and are not designed for educational purpose. They have the same educational limitations as in the case of professional firewalls.

On the other hand, compared to traditional firewall approaches such as IPtables and IPfw, the simulator tool approach is more adequate for developing educational lab exercises and allows providing the students with more advanced educational firewall functions, such as writing low level filtering rule, packet content inspection, and consistency and efficiency verification of filtering rules. In addition, the simulator tool approach provides more friendly interaction between educators and students.

4 Design Considerations

Firewalls control the access into and from networks based on a set of filtering rules which reflect and enforce the organization’s security policy. Within a network, the firewall is typically the first filtering device that encounters packets that attempt to enter an organization’s network from the outside, and is typically the last device to see exiting packets. It is the firewall’s job to make filtering decision on every packet that crosses it: either to let it pass, or to drop it.

There are a number of basic topics about network packet filtering and inspection that should be taught when offering a security course on firewalls:

- Basic packet filtering
- ICMP traffic filtering
- Common standard services (HTTP, FTP, SMTP, POP3, Telnet, and DNS) filtering
- Nonstandard services filtering
- Packet content inspection
- Stateless and stateful firewalls
- Consistency and efficiency verification of filtering rules
- Filtering rules order management

When designing and implementing an educational firewall simulator tool, the above basic topics should be taken into consideration. The simulator tool is expected to
offer advanced functions that are not usually available in common professional firewalls. In fact, the simulator tool should provide its users with the adequate functions that allow implementing basic and advanced packet filtering rules for both standard and nonstandard network traffic and services, using preferably graphical user-friendly interfaces. In addition, the simulator tool’s users should be able to verify the consistency and efficiency of their defined filtering rules as well as viewing the contents of the stateful tables of the established network sessions. Moreover, the simulator tool should allow its users to easily modify the order of the filtering rules to adequately reflect the security policy under consideration.

Other topics, such as application gateway firewalls (Proxy), Virtual Private Networks (VPN), firewall secure network architectures, are also usually covered by a security course on firewalls. However, the current version of the firewall simulator tool does not cover them. A future version of the tool can include more educational functions to cover these topics.

Based on the above design considerations, the educational objectives of Edu-Firewall tool are outlined as follows:

- Provide an educational solution that helps students to improve their hands-on security skills on firewalls.
- Offer students the adequate means to deal with advanced firewalls security features.
- Provide students with a user-friendly simulated environment for firewall configuration, and for writing low level packet filtering rules.
- Provide educators with means to evaluate students’ hands-on security skills on firewalls.

5 Educational Functions of Edu-Firewall Tool

Edu-Firewall tool can be used by security educators as well as by students to write basic and advanced filtering rules for given security policies, and to optimize the filtering rule set. Edu-Firewall tool allows students to select security policies from the available list which has been created by the educators, and then write the corresponding filtering rules. On the other hand, Edu-Firewall tool allows then the educators to evaluate the students work. Depending on the complexity of the selected security policy, the corresponding filtering rules can be simple or complex to implement. Edu-Firewall tool uses web interfaces that include a set of functions to allow the students implementing both simple and advanced filtering rules. Edu-Firewall tool’s educational functions cover the main topics taught in a security course on firewall concepts. The offered educational functions of Edu-Firewall tool allow performing mainly the following tasks:

- Define advanced TCP filtering rules by freely setting all TCP header fields, including the TCP flags.
- Define advanced ICMP filtering rules by freely setting all ICMP header fields.
- Define advanced UDP filtering rules by freely setting all UDP header fields.
- View the contents of the TCP, ICMP, and UDP stateful sessions table.
- Verify the consistency and efficiency of the filtering rules.
- Optimize the order of the filtering rules.
- View the logs of filtered packets.

The following subsections describe briefly the main Edu-Firewall tool’s educational functions.

5.1 Basic Packet Filtering

Basic firewall packet filtering is the selective passing or blocking of packets as they pass through a network interface. The most often used criteria that packet filtering uses when inspecting packets are source and destination IP addresses, source and destination TCP/UDP ports, TCP flag bits, type and code fields in an ICMP header, and the protocol field of the Layer 4 header. For example, to filter out all Ping traffic coming to a network, the firewall simulator tool should allow implementing filtering rules that block all incoming ICMP echo request packets (Type=8 and Code=0). The following filtering rule reflects the above security policy: (Direction = Incoming, Source IP = Any, Destination IP = Any, Protocol = ICMP, Type = 8, Code = 0, Action = Deny).

5.2 Packet Content Inspection

Packet content inspection is a form of network packet filtering that examines the data part (Payload data), and possibly also the headers of a packet as it passes a firewall. Usually, packet content inspection searches for protocol non-compliance, viruses, spam, intrusions or predefined criteria to decide if the packet can pass or if it needs to be routed to a different destination.

Usually, packet content inspection process uses a set of signatures, commonly created by the firewall administrator. For example, you might want to prevent your network’s users from receiving emails from a specific email address.

5.3 Nonstandard services filtering

Standard services run usually on standard ports. For example, the standard ports for HTTP and FTP services are 80 and 21, respectively. Firewalls include usually predefined rules to filter standard services, and are unable to filter nonstandard services unless the user provides the firewall with the TCP or UDP ports of the nonstandard services. In practice, this is achieved by creating a new service profile for the nonstandard service and specifying its corresponding TCP or UDP port number.

5.4 Consistency and Efficiency Verification of Firewall Filtering Rules

The consistency and efficiency of a firewall strongly depends on the ability of the administrator to develop well defined and coherent filtering rule set, and to be able to continuously clean and verify the correctness of the rules. It is important to mention that in cases where there are dozens or hundreds of filtering rules, inconsistent and inefficient filtering rules with anomalies might not be easy to spot. Hence, checking a large set of filtering rules for inconsistencies and inefficiencies is difficult and prone to errors when it is done manually and in an ad-hoc manner. Thus, automated tools are required to assume such a task. However, to our knowledge, currently there is no professional firewall that integrates strong filtering rule consistency and efficiency verification capabilities.
For example, Juniper firewalls (Table 1) provide a very limited online command to identify inconsistent and/or inefficient filtering rules.

Edu-Firewall tool allows verifying the following three types of inconsistency and inefficiency within the filtering rules.

**Shadowing:** A rule is shadowed by a preceding rule if it is a subset of the preceding rule; and the two rules define different actions. That is, an upper rule shadows a lower rule, when the upper rule matches all the packets that are also matched by the lower rule, such that the lower rule will never be reached by the firewall. To correct this situation, simply remove one of the filtering rules, or reverse the order of the two filtering rules, putting the more specific one (shadowed rule) first. The following two rules are example of shadowing rules:

**Rule #1:**
- **Direction = Incoming**, **Source IP = Any**, **Destination IP = Any**, **Protocol = TCP**, **Source Port = Any**, **Destination Port = 80**, **Action = Allow**.

**Rule #2:**
- **Direction = Incoming**, **Source IP = 192.168.3.30**, **Destination IP = Any**, **Protocol = TCP**, **Source Port = Any**, **Destination Port = 80**, **Action = Deny**.

**Redundancy:** A redundant rule performs the same action on the same packets as another rule such that the removal of it would not affect the operation of the firewall.

**Contradictory:** two rules have the same parameters except the action is opposite.

6 Edu-Firewall Tool’s Implementation

Edu-Firewall tool is a web application, and has been implemented using Java language as a programming language; specifically, JSP (JavaServer Pages) using NetBeans IDE software as a development environment. In addition, mySQL has been used to create the Edu-Firewall tool’s database of security policies and filtering rules.

6.1 Edu-Firewall Tool’s Web Interfaces

Edu-Firewall tool offers two main web interfaces. The first web interface is dedicated for the instructors to create and update security policies (Figure 1).

The second web interface is for students to use the educational firewall functions offered by Edu-Firewall tool. That is, after logging to the tool, the student will be redirected to a web page from which he/she can select a security policy scenario. Then, the student is asked to enter the adequate filtering rules for the selected security policy. This requires creating, editing and ordering the filtering rules. The student will also be able to verify the consistency and efficiency of the entered filtering rules. All the added filtering rules will be saved in the database and then will be evaluated by the instructors. Before defining the filtering rules, the student should configure the network interfaces through the Interface configuration web page.

**Network interface configuration:**

The Interface configuration web page is designed to set the Ethernet network interfaces. Students can configure up to two network interfaces. This page allows the student to specify the network IP address, the interface IP address, the network mask address, and the Dynamic Host Configuration Protocol (DHCP) IP address range (Figure 2).

**Filtering rule definition:**

This web page allows students to write basic and advanced filtering rules for standard and nonstandard services, and deep packet content inspection. Depending on the selected security policy, the student is requested to enter the appropriate filtering rules by specifying the values of the packet’s fields that should be inspected by the firewall.

For each filtering rule, the simulator tool shows also the direction of the network traffic (incoming or outgoing) that will be filtered by the firewall. Figure 3 shows the list of the TCP packet’s fields that can be set when writing a filtering rule.

**Packet deep inspection:**

Edu-Firewall tool’s Deep Inspection web interface is designed to provide the application layer inspection for the most prevalent Internet-facing protocols, such as HTTP, FTP, SMTP and POP3. In this web interface, the student will specify the attack name, attack context and attack pattern for packet deep inspection (Figure 4).

**Inconsistency and inefficiency verification:**

Edu-Firewall tool’s rule verification web page allows students to verify the consistency and efficiency of the created filtering rules. Practically, this function allows detecting shadow, contradictory and redundancy filtering rules (Figure 5).

7 Evaluation of Learning Outcomes

Several hands-on lab exercises on firewall have been offered in our Network Border Control course (SECB358) during the last three academic years. SECB358 course teaches students mainly firewall concepts and technologies, and packet filtering and inspection techniques. The exercises require the use of Edu-Firewall tool by the students.

This section discusses the effect of introducing Edu-Firewall tool during the hands-on lab exercises on the achievement of the SECB358 course’s outcomes (COs). SECB358 course has five COs as shown in Table 2. Since SECB358 course is an advanced course in information security, the outcomes have been selected carefully to reflect the top three levels in the bloom’s taxonomy of cognitive domain (analysis, synthesis, and evaluation).

After creating the course outcomes, 10 course topics were identified and mapped to the course outcomes. Four assessment tools are also selected to assess the achievements of COs including quizzes, exams (midterm and final), lab reports, and term project.
Table 2: Mapping the course outcomes to Bloom's Taxonomy

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Level of Bloom's Taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO1: Describe TCP/IP protocols and network services.</td>
<td>Analysis (4)</td>
</tr>
<tr>
<td>CO2: Identify common security threats.</td>
<td>Analysis (4)</td>
</tr>
<tr>
<td>CO3: Configure personal firewalls, network firewalls, and VPNs.</td>
<td>Synthesis (5)</td>
</tr>
<tr>
<td>CO 4: Implement firewall filtering rules for different network architectures and services.</td>
<td>Synthesis (5)</td>
</tr>
<tr>
<td>CO 5: Evaluate different types of network architectures.</td>
<td>Evaluation (6)</td>
</tr>
</tbody>
</table>

To assess the course outcomes, we follow the course assessment process adopted by our institution. A nominated course coordinator assembles a course committee that includes all the lecture and lab instructors teaching the course in a given semester. During the first week of the semester, the course committee meets to decide on the assessment tools that will be used to assess the COs. They also decide on the corrective actions that will be applied to address the recommendations from the previous assessment cycle. Throughout the semester, the course committee applies the assessment tools to collect assessment data. By the end of the semester, the collected assessment data are mapped to the COs. The achievement level of each CO is then calculated in terms of mean and standard deviation using (1) and (2).

\[
\mu (CO_i) = \frac{\sum \mu_i \times n_i}{\sum n_i} \quad (1)
\]

\[
\sigma (CO_i) = \sqrt{\frac{\sum \sigma_i^2 \times n_i}{\sum n_i}} \quad (2)
\]

where \(\mu_i\) and \(\sigma_i\) denote respectively the normalized mean, and standard deviation of the students’ marks when assessment tool \(i\) is used, and \(n_i\) denotes the number of students. For example, if three quizzes and two final exam questions are used to assess CO\(_n\), the normalized mean and standard deviation of the students’ marks are calculated separately for each tool, then (1) and (2) are used to calculate the achievement level for CO\(_n\). After calculating the achievement level for each CO, the course committee meets again to discuss the assessment results and decide on the needed recommendations to address any discovered shortcoming. To close the assessment cycle, the course committee also discuss the effectiveness of the corrective actions applied during the semesters on the new assessment results.

During the 2008/2009 and 2009/2010 academic years, students enrolled in SEC358 course were not offered Edu-Firewall tool during the hands-on lab exercises. Only the firewall theoretical concepts were described during the lecture time. However, starting from fall 2010, the course committee decided to offer Edu-Firewall tool during the hands-on lab exercises as a corrective action to improve the COs achievement levels and improve the students hands-on skills on firewalls. Three quizzes are used to compare the achievement of the COs before and after the introduction of Edu-Firewall tool. These quizzes are directed mapped to CO3 and CO4 outcomes. The grades of the students in the three quizzes are measured, normalized, and then aggregated using (1) and (2) to calculate the achievement level of the two outcomes.

7.1 Assessment Results

Figure 6 shows the students’ average grades for the three quizzes used to evaluate the students’ comprehension of firewall configuration and filtering rule implementation. It shows clearly that starting from 10/11 academic year, the total average grade has started improving. This is mainly due to the fact that the offered Edu-Firewall tool allowed students to better analyze and assimilate the firewall concepts learned from the lecture. The students have learned better with Edu-Firewall tool which had a positive effect on their performance. For example, in case of Quiz 2, introducing Edu-Firewall tool improved the average student grade by 9% from 0.68 to 0.74 and maintained the improvement for the following two academic years.

Figure 7 illustrates the achievement of the two course outcomes (CO3 and CO4) for five consecutive academic years from 08/09 to 12/13. It shows an improvement in the two outcomes achievements level after introducing Edu-Firewall tool. For example, the introduction of Edu-Firewall tool in 10/11 academic year improved the achievements levels by 3% and 9% for CO3 and CO4 respectively compared to the achievement levels in the year before. It is important to indicate that the introduction of Edu-Firewall tool improved slightly the achievement level of the outcome CO3 since the VPN concept is not covered by the tool. In contrast, the achievement level of the outcome CO4 has been improved well since the tool offers a set of educational functions covering all the topics related to this outcome, specifically implementing basic and advanced firewall filtering rules.
8 Student’s Satisfaction

An anonymous questionnaire was administered to 120 students who used Edu-Firewall tool, to measure their satisfaction level and collect their feedback regarding the tool. The results of the questionnaire showed that about 93% of the students who answered the questionnaire believed Edu-Firewall tool to be useful and helped them better understand the underlying theoretical concepts associated with firewalls and packet filtering (Table 3). The questionnaire also revealed that about 80% of the students agreed that the tool helped them to develop further their hands-on skills, and about 91% of the students would strongly see similar simulator tools offered for other security topics.

<table>
<thead>
<tr>
<th>Question</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you think it is easy to create and manage filtering rules using Edu-Firewall tool?</td>
<td>5% Disagree, 10% Neutral, 23% Agree, 62% Strongly agree</td>
</tr>
<tr>
<td>Do you think Edu-Firewall tool offers to you the expected basic firewall features?</td>
<td>3% Disagree, 8% Neutral, 10% Agree, 79% Strongly agree</td>
</tr>
<tr>
<td>Do you feel that Edu-Firewall tool helped you to develop further your hands-on skills?</td>
<td>7% Disagree, 13% Neutral, 13% Agree, 67% Strongly agree</td>
</tr>
<tr>
<td>Do you feel you understand the firewall theoretical concepts better after using Edu-Firewall tool?</td>
<td>2% Disagree, 5% Neutral, 8% Agree, 85% Strongly agree</td>
</tr>
<tr>
<td>Do you think you will recommend Edu-Firewall tool to others?</td>
<td>5% Disagree, 5% Neutral, 3% Agree, 87% Strongly agree</td>
</tr>
<tr>
<td>Would you like to see similar simulator tools offered for other security topics, such as intrusion detection?</td>
<td>5% Disagree, 4% Neutral, 18% Agree, 73% Strongly agree</td>
</tr>
</tbody>
</table>

Table 3: Student satisfaction questionnaire results

9 Conclusion

It is necessary that students enrolled in information security programs acquire solid hands-on skills on firewalls configuration and filtering rules. This paper described an educational web-based firewall simulator tool to help students learn the intricacies of writing firewall filtering rules to filter and inspect network traffic. The simulator tool offers a set of educational functions that are not commonly available in professional firewalls. The tool is designed to be used as a part of an undergraduate or graduate level course on firewalls.

As a case study, the impact of using the simulator tool on the students’ performance in terms of achieving the course outcomes is also discussed. The assessment results showed a significant improvement in the achievement level of related course outcomes. Overall, the students have learned better with the tool which had a positive effect on their performance.

Future versions of the tool will include several uncovered educational firewall functions, such as VPN network configuration. In addition, to effectively evaluate the students’ defined filtering rules, the network traffic that will be filtered and inspected by the simulator tool will be captured directly from the Internet or from LAN networks.

10 Acknowledgement

The authors acknowledge the contribution of the following students in the implementation of the web interfaces of Edu-Firewall tool: Alia Sultan AlNuaimi, Khawla Salem Al-Mazroee, Latifa Rashed Al-Ketbi, and Shamsa Rashed Al-Neyadi.

11 References


<table>
<thead>
<tr>
<th>Firewall</th>
<th>Juniper Networks Version: 6.2.0r5.0 (firewall + VPN)</th>
<th>Jetico Personal Firewall v.2.1.0.10</th>
<th>Cisco ASA 5520 Version: 7.0 (7)</th>
<th>Proventia Version: 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create new services</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Rules parameters</td>
<td>Multiple Yes</td>
<td>Multiple Yes</td>
<td>Multiple Yes</td>
<td>Multiple No</td>
</tr>
<tr>
<td>IP address</td>
<td>Single Yes</td>
<td>Single Yes</td>
<td>Single Yes</td>
<td>Single Yes</td>
</tr>
<tr>
<td></td>
<td>Range Yes</td>
<td>Range Yes</td>
<td>Range Yes</td>
<td>Range Yes</td>
</tr>
<tr>
<td></td>
<td>Discrete No</td>
<td>Discrete No</td>
<td>Discrete No</td>
<td>Discrete No</td>
</tr>
<tr>
<td></td>
<td>Network Yes</td>
<td>Network Yes</td>
<td>Network Yes</td>
<td>Network Yes</td>
</tr>
<tr>
<td>Port number</td>
<td>Standard/Default</td>
<td>Single port</td>
<td>Standard/Default</td>
<td>Single port Port range</td>
</tr>
<tr>
<td>Actions</td>
<td>Allow/Deny/Reject</td>
<td>Accept/Reject/Continue</td>
<td>Permit/Deny</td>
<td>Drop/Protect/Ignore/Monitor</td>
</tr>
<tr>
<td>Protocols</td>
<td>Wide range of supported protocols</td>
<td>Wide range of supported protocols</td>
<td>TCP/UDP/ICMP/IP</td>
<td>Wide range of supported protocols</td>
</tr>
<tr>
<td>Deep inspection</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Writing low level rules</td>
<td>Not supported</td>
<td>Partially supported</td>
<td>Not supported</td>
<td>Partially supported</td>
</tr>
<tr>
<td>Default rules</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Consistency and</td>
<td>Partially supported</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>efficiency verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Setting TCP flags</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Stateful/Stateless</td>
<td>Stateless</td>
<td>Stateful</td>
<td>Stateful</td>
<td>Stateful</td>
</tr>
</tbody>
</table>

Table 1: Limitation of professional firewalls

![Figure 1: Web interface for creating and updating security policies](image1)

![Figure 2: Web interface for network interface configuration](image2)
Figure 3: Web interface for filtering rule definition

Figure 4: Web interface for packet deep inspection

Figure 5: Web interface for filtering rule consistency and efficiency verification
A Multi-tier Client-Server Project Employing Mobile Clients

Chris McDonald
School of Computer Science and Software Engineering
The University of Western Australia
Crawley, Western Australia, 6009
chris.mcdonald@uwa.edu.au

Abstract

The current generation of Computer Science students are far more likely to engage with computer networking through their own mobile, wireless devices than they are using wired, desktop computers. Traditional approaches to teaching computer networking evolved when the Internet was composed of fixed wired infrastructure, and this historical background still forms most of the material in contemporary textbooks on computer networking. Today’s students have strong expectations that their computer networking units will have a significant focus on the networking devices and applications that they use daily - increasingly mobile and wireless.

This paper describes a client-server networking project which requires students to design, implement, test, and analyze a client-server software architecture, using both desktop computers, and handheld, mobile, wireless devices. The application domain of the project is location prediction using only WiFi beacon frames – an application familiar to most students, but one about which they initially had little curiosity. The paper then reflects on the many lessons learnt from this project, both from the perspective of the students and the professor.

Keywords: Mobile and Wireless Networks, Location Prediction, Computer Science Education

1 Introduction

It can be argued that computer networking continues to be the field of computing making the greatest impact on our daily lives, with the Internet becoming the ubiquitous carrier of global applications such as user-contributed videos and music, instant messaging, location-aware mapping and applications, and the myriad of competing and often contradictory information sources. Our students’ almost insatiable desire for access to the Internet often pushes aside the curiosity to understand the technical aspects of how it all works, and even why it all works. Burd et al. (2012) present extensive arguments as to why the rise, role, and popularity of mobile computing throughout society should be strongly supported by a re-examination and update of relevant areas of the Computer Science curriculum.

Contemporary university students, collectively described as the M-generation, Generation-Y, and Millennials, are more likely, or wish, to access many Internet services using handheld and mobile devices, including laptop computers, smartphones, and tablet computers, as reported by Junco & Mastrodicasa (2007), Oblinger & Oblinger (2005).

The rapid growth in mobile and wireless computing, and its emerging role in Computer Science Education, is easily witnessed by the number of recent publications in the literature. Many of these publications, particularly those form as “long ago” as 2008, have reported on the application of using mobile devices to teach design and human-centric computing, often focusing on the challenges of the devices’ small form-factor. More recent papers have described the introduction of mobile phone application development into Computer Science curricula, both at university and college level and, more significantly, into the general computing curricula at high-school and (the US) K-12 levels. Specifically, the use of mobile devices and sensors are often used in university and school Outreach Programmes, to motivate prospective university students. Many newer publications focus on the use of existing application development environments, mostly for the Android platform, but a few for Apple’s iOS, and often employ mobile games development or use of the devices’ inbuilt 3D, light, and sound sensors (Bayzick et al. 2013, Dabney et al. 2013).

More relevant to this paper, have been a number of publications that focus on the use of mobile and wireless devices as general purpose computers, and as vehicles to study established areas of Computer Science such as Operating Systems and Computer Security (Andrus & Nieh 2012, Riley 2012, Guo et al. 2013).

2 Motivation

For current students in Computer Science units the Internet’s “last-mile” problem is being solved, not through traditional fibre or copper connections to the home, but by near-ubiquitous wireless access. Consequently, students undertaking units in computer networking today are expecting their units to contain, if not focus on, details of wireless and mobile networking developed this century, and not to remain focused on the traditional, often inaccessible, wired backbone infrastructure and their protocols, developed in the 1970s.

While it remains necessary that Computer Science units explain the networking fundamentals of data corruption and loss, switched versus packet-based transmission, layered models, and the roles and responsibilities of numerous protocols at
all layers of the networking stack, the always-on, instant-connection nature of broadband connections and mobile technologies pushes educators to talk about the detailed technical infrastructure in place. Traditional topics, such as Shannon’s communication theory, error correcting codes, and algorithms drawn from graph theory, can no longer hold pride of place in the networking classroom.

To maintain students’ engagement and, more importantly, to increase their understanding of the design and implementation of modern computer networks, it is now necessary to convey many of the fundamental concepts by drawing upon mobile and wireless networking examples. This paper presents our recent successes with student projects developing a location prediction application using the wireless communication facilities of modern mobile devices, such as Apple iPods.

An increasing number of mobile handheld devices, described as smartphones, provide Internet connectivity through wireless ethernet (WiFi) and as through cellular phone infrastructure (in Australia, 3G and 4G). Higher-end models also provide GPS receivers to provide accurate location information. However, the results that can be determined by GPS alone fail or degrade in a number of environments, including urban canyons and the interior of buildings. This is precisely where WiFi can exhibit great advantage – in environments containing many WiFi access points, such as on a university campus. Most modern smartphones and their operating systems support hybrid approaches to localization, combining GPS signals, the known locations of fixed-telephony infrastructure (mobile phone towers), and explicitly collected or crowd-sourced information including the MAC addresses and signal strengths of observed WiFi access-points.

Students are very familiar with the location prediction software provided by the operating systems, and third-party applications, on their smartphones. Depending on the types and strengths of signals available to the device, location determination generally employs the device’s GPS receiver, or a combination of GPS and WiFi signals. Devices report results and changes when time or distance intervals are exceeded. Location determination using WiFi signals emitted from known WiFi access-points usually requires access to a very large database of pre-captured information, such as that provided by the mapping services of Google, Apple, and Skyhook Wireless. However, when used alone, location prediction using only WiFi often only claims an accuracy to 150 metres in urban areas, as prediction is obviously limited by the number and accuracy of samples taken.

Wikipedia (Wikipedia 2012a) describes geolocation as “the identification of the real-world geographic location of an Internet-connected computer, mobile device, website visitor or other... Geolocation may refer to the practice of assessing the location, or to the actual assessed location, or to locational data.”

This paper reports on projects undertaken in successive years, by undergraduate students to form part of their assessment in their core computer networking unit. In these projects students implemented a form of geolocation by employing the WiFi signal information received by handheld mobile devices. While the problem domain, and its uses, are very familiar to our students, the projects have demonstrated areas of our teaching and of our students’ knowledge that could do with improvement.

3 Background

The Computer Networks unit at The University of Western Australia is offered to students in their third undergraduate year, students undertaking bachelor degrees in Computer Science, Software Engineering, Electronic Engineering, and Business, and to students undertaking a “conversion” Masters by Coursework degree, whose first degree is frequently other than computing or information technology. While most undergraduate students have taken other systems-focused units, such as Operating Systems and Security and Privacy, most of the unit’s Masters by Coursework students have arrived with very diverse backgrounds, which has caused many challenges in this unit. In recent years the unit has received enrolments of between 55 and 160 students.

The unit comprises one eighth of a year’s work, and is assessed with a practical project contributing 40% and two examinations contributing 60%. In recent years, the practical projects described in this section, ran for 5 weeks each and were undertaken in self-selecting groups of two or three students.

The goal of each student project was to design, implement, and analyse a location-prediction application using a multi-tier client-server architecture. Students were to develop software forming part of a mobile application, to execute on Apple iPod Touch devices, and a simple server running on a desktop machine (typically running Linux) which communicates with a provided black-box server using a pre-defined protocol. As well as developing working software, students were also required to to analyse and report on the observed effectiveness of their software. The project was designed to improve students’ understanding of the physical wireless environment, IEEE802.11 wireless protocols, and to assess their understanding of Transport Layer networking protocols, and their ability to analyse and report on their observations. While the project was not designed to teach skills in mobile architectures or the programming of mobile or embedded devices, the project also provided an initial exposure to these topics for most students.

In some schools, such a project may appear under-specified, as very little explicit information was provided as to how students should undertake their mobile and server application design and evaluation. Students had already completed four weekly closed laboratory sessions which had introduced them to the data-link layer, routing, and wireless network protocols. The cnet network simulator (McDonald 1991, 2009) had, in particular, encouraged them to experiment with the fundamentals of wireless signal transmission and reception, typical transmission distances, and the handling of frame collisions. However, the use of cnet for many years had clarified, in the author’s mind, that too many students were perceiving the simulator, and its ease of use, to be an accurate reflection of programming and testing mobile applications. While simulation certainly still holds a deserved position in the teaching of computer networking, the emergence of mobile and wireless devices means that we should now offer our students a richer experience.

4 Project Methodology

The complete project was undertaken over five weeks, by teams of two or three students. The project employed a small number of distinct software applications and network protocols, and students’
solutions required two primary phases to achieve a successful outcome.

### 4.1 The provided photoserver software

The author developed a simple server, named the photoserver, to host and deliver GIF-format images that had been taken using a smartphone. For our project, about 300 equally-sized images from 60 locations were collected from a region of about 400x300 metres, near our Computer Science building. The latitude, longitude, and compass direction faced (when taken) of each image were manually recorded although, today, many digital cameras and smartphones can themselves provide this information.

Students were provided with details of how to contact the photoserver, including its IPv4 network address and TCP-based port with which it could be contacted. At the conclusion of the project, interested students were provided with the source code to the server (although similar implementations are readily available over the Internet). Student also received details of the simple TCP/IP text-based protocol used to communicate with the server. The protocol defined a small number of commands (verbs) understood by the photoserver, and its valid responses and error conditions. In essence, the commands and responses were similar to many existing text-based standards, such as for FTP or TELNET. Commands could request that the photoserver report the geographical region that it was “managing”, and to request the downloading of images from specific locations (latitude, longitude, and heading) within the region.

While students received details of the photoserver’s protocol, they were not provided with a pre-built client-side application programming interface (API), in any specific programming language, with which to access the photoserver. This enabled students to write their geoserver software (described later) in their choice of programming language, and required them to implement the necessary bidirectional communication using their language’s Berkeley socket APIs and I/O functions or methods. Most students adopted their preferred language taught and used in their earlier units – Java, C, or Python in decreasing order of “popularity”.

These design choices, in themselves, provided three important lessons to students – that clients and servers communicating over the Internet need not be written in the same programming language, that text-based protocols facilitate communication between disparate operating systems and hardware, and that text-based protocols are generally sufficiently efficient for such simple tasks. Because the photoserver’s protocol was text-based, students could experiment with it using existing command-line programs, such as netcat, and then have their own programs mimic these actions by sending text-based commands, across a socket, and receiving and parsing responses.

The photoserver software was also deliberately placed behind a firewall that prevented it being accessed, directly, from the students’ mobile devices. This enforced the project’s requirement that students develop both a mobile device application and a desktop server application (described shortly). Sufficiently enthused students could have added more photographs from across the whole campus, but none of our students even asked why the chosen region was so limited – it is likely that none ventured outside of this region.

### 4.2 The students’ mobile application

The second part of the project required students to walk around the region, capturing WiFi beacon signals from the university’s WiFi network. Our university has an extensive WiFi network, covering nearly all of its campus, with about 50 of its 360 wireless access-points “visible” in the physical region of our project. The students were provided with 8GB Apple iPod Touches for the duration of the 5-week project, and with code for a mobile application to extend and run on the devices. The application’s provided code captured signals from the device’s wireless network interface, and managed a simple multi-screen interface; see Figure 1. The interface enabled the students to set the network address of their geoserver software, displayed a map of the campus on which locations of previously captured WiFi beacons could be identified, a screen to display images fetched from the photoserver (via their geoserver), and a text-based logging/debugging screen.

The iPod’s mobile application was written by the author in Objective-C, and made calls to Apple’s iOS library to provide the graphical interface and to gather information about captured WiFi beacon frames. Our students gain no direct experience with Objective-C throughout their degree, although some learn it through their personal interest in programming Apple iOS and OS-X devices. Learning a new programming language is not an objective for our Computer Networks unit, and so students were encouraged (permitted) to develop all of their own code on the iPods in ISO-C99, a language they had previously learnt in earlier years, and had also used extensively in a recent Operating Systems unit.

Students were provided with a standard Makefile to compile and link the provided Objective-C code and their own C code, to build the iPod application, and a simple ssh/scp command sequence to push their applications to the mobile devices over a USB connection. The number of students enrolled in the unit introduced a difficulty not initially anticipated – despite working in groups, almost every student wished to undertake some development of their team’s mobile application. Our school could not provide access to sufficient Apple desktop machines to meet this demand, and so the author developed a network-based cross-compiler enabling students working on the school’s Linux desktops to have their mobile applications compiled and linked on Apple hardware. Although the project required some investment in and commitment to Apple’s iOS platform, it could similarly be developed and undertaken using Android on suitable devices.

The student written C code in the mobile application responded to button presses on the device’s screen. In the first phase of the student’s data collection, students scrolled the on-screen map to align crosshairs with their actual current location, and then pressed a button to provide their code with triples of the most recently captured WiFi information, including each access-point’s MAC address, signal set identifier (SSID), and the relative signal strength of the arriving signal (its RSSI value). In combination, this information forms a wireless fingerprint of the location where it was captured. The information was sufficient for the students’ projects to determine the approximate physical location of their mobile device, and for that information to be further used to request a photographic image of the scene that should be observable from that location – a poor man’s Google StreetView.
Students employed a variety of strategies to process their captured data, with decisions taken dependent on their related choices for their geoserver’s design. Nearly all student teams transmitted their captured data back to their geoserver as formatted ASCII text, often with one WiFi reading per line. It is unclear if students generally recognized the benefits of, and chose to use, a text-based format based on the early discussions in class sessions, or if they were simply mirroring the provided protocol between the photoserver and their geoserver. Most teams, initially, chose to employ TCP stream connections to their server software, and sent the captured information from a single location as a sequence of lines. Other teams packed the same information into a single UDP datagram, resulting in their need to also add sequence numbers and their tracking to their protocols. The challenges introduced by lost data while using UDP datagrams led a number of teams to adopt TCP streams, instead.

4.3 The students’ geoserver software

The students’ second task was to design and implement their geoserver, software sitting midway between the provided photoserver and their mobile application, and acting as a proxy for requests, results, and errors. As the photoserver was firewall-ed from our campus wireless network, and thus from the mobile applications running on mobile devices with known DHCP-allocated addresses, all communication had to, at least, pass through the geoservers. Students’ geoservers ran on either desktop machines or on virtual servers in our Computer Science building. Ironically, during testing, students “in the field” often used their mobile phones to communicate (by phone) with other students controlling their team’s geoserver application on the desktop machines.

The primary role of the geoserver was to determine the apparent location of the mobile device, based on the WiFi signal information that it was currently receiving from the mobile device. This concept of an apparent location was central to the project, but was initially misunderstood by many students.

Although our campus is well covered by over 360 access-points, the project did not require knowledge of their actual location. Instead, the fingerprints collected during the project’s first phase were used to determine each access-point’s apparent location – holding the location beneath the mobile interface’s crosshairs as the ground-truth, and then predicting where each access-point’s location was “around” this point.

With sufficient walking and sampling of the region, each access-point is observed several times, with its WiFi beacon frames arriving at each claimed location with a variety of signal strengths. At the end of the project’s first phase, each team had used their captured data to determine an apparent location for each access-point, but each particular access-point’s apparent location varied, depending on from where, and how many times, it was observed. Student teams employed a variety of algorithms, including self-described “intuitive” algorithms, to determine apparent locations from sampling locations. The most common of these were traditional triangulation and trilateration (Wikipedia 2012b,c). The first of these is the simpler, and can be easily derived anew from basic geometry – in fact one team claimed to have discovered it, and named it after themselves. Triangulation, although simple, does not take into account the strength of arriving signals and, thus, places all sampled points equidistant from the sample centre.

More complex is trilateration, which uses the arriving signal strength to represent a logarithmic inverse of the distance between the sampling centre and access-point; see Figure 2. As the signal strength of beacon frames from access-points dissipates according to the inverse-square law, trilateration provided more consistent results when predicting new locations with respect to previously visited ones.

Some student teams employed techniques that were reliant on them being able to find the wireless footprint most-similar to ones previously captured, and two teams even attempted to develop a detailed grid of the whole (assumed) environment, to a resolution of 5 metres. These latter two approaches often led to unsuccessful projects (ones not being able
to accurately predict current locations, or to deliver a photo from the predicted location), as the techniques were reliant on very repeatable observations. Such attempts were quite disappointing, as the students could have easily observed that, even while standing still, received signal strengths vary by amounts that would displace their predictions by as much as 30 metres. The clear inference is that students did not perform enough testing of their applications.

With respect to the traditional networking requirements of the project, students were exposed to exactly the same details about the Berkeley sockets API, client-server connection, and communication patterns as they would have been had the project simply been undertaken only on desktop machines connecting to the wider Internet using wired infrastructure. However, what this project specifically introduced students to, was the need to develop robust software that did not crash in the presence of failure. Client-server software using wireless networks, even using wireless networks in laboratories or libraries rarely fails, and typically exhibits fast, regular, communication patterns – for the basic reason that its devices are rarely mobile.

This project not only required the students’ code to make connections on-the-fly, but to detect frequent connection failures. Moreover, students had to design their software to ensure that queued communication was eventually transmitted. Students commented that they had never before had to write software that would experience failure, with I/O operations that could fail and require retrying. In our earlier units, such as Operating Systems, attention focuses on capturing and reporting the success or failure of read and write disk-based operations, but not (if ever) are students required to consider retrying failed operations, perhaps asynchronously. This is a consequence of Operating Systems being presented at (only) second-year level. There is a need for earlier units to more strenuously emphasize and assess the robustness of students’ software and for units, such as Computer Networks, to set student projects in environments which do experience network failures.

Some of the other lessons learnt were quite surprising, and demonstrated the diversity of students’ backgrounds – more so than their ability to cope with certain problems. As stated in the previous section, the requirement that only the apparent location of access-points was required was foreign to some students, who insisted on knowing the true location of access-points (which we did not have nor seek). Students initially appeared to lack confidence in being able to develop results in the project’s first phase, and to then treat these as axiomatic for the second phase. Moreover, students learnt that the nature of wireless transmissions can be quite fickle, and that the often seen textbook explanations and simulations of signal propagation, in perfect circles, is never seen in practice (Kotz et al. 2004).

Students also had some difficulty using latitudes and longitudes as a Cartesian reference system, and mapping these to and from pixel-based coordinates on maps. We have no explanation for this difficulty, other than to report that most students had not undertaken an earlier unit in Computer Graphics, and that the problem sometimes arose because we are in the southern hemisphere, where latitudes increase (but become more negative) while moving southwards, or “down” the device’s screen.

Students even discovered some other unique problems, totally by accident. One team found that their application kept crashing when walking through a certain area on campus, but worked well elsewhere. Extensive testing amongst the students revealed that a particular private WiFi access-point had used a full 32-byte SSID and was therefore not delivering the expected null-byte to signal the end of its SSID name. This sort of error might never have turned up in normal testing, and certainly never in simulation, but may have caused a similar application to crash when used in practice.

Figure 2: Using trilateration to predict the location of WiFi access-points (Wikipedia 2012c).

A few student teams demonstrated creativity by choosing to plot the apparent locations of access-points on a campus map, using user-defined layers in either Google Maps or Google Earth. While all on-campus access-points are located inside buildings or under overhanging balconies, a number of access-points appeared to be positioned in the middle of grassed areas, and in gardens. While physically unrealistic, this eventual placement is simply a result of the fact that WiFi signals are absorbed, reflected, and refracted as they pass through most building materials and vegetation. Even some atmospheric conditions, such as rain and high humidity, can have observable effects on the strength of arriving WiFi signals, and some teams’ results would have been skewed if conditions on the days they sampled, were very different to those on the days they employed their predictions.

5 Student experiences

The goal of this project was to increase and assess students’ understanding of the physical environment of wireless networks, transport layer protocols, and their ability to analyse and report on the observed effectiveness of their software. Although students were not formally surveyed, it is very clear from the many comments and questions raised on our unit’s online forum, that the students enjoyed the nature of the project, and its use of mobile devices as, simply, mobile computers. They also particularly enjoyed devising approaches to a problem that appears innately solvable, but one that is so influenced by many external factors that only approximate solutions are possible.
6 Summary

This paper has motivated and described in detail a project suitable for third or later year undergraduate students undertaking a unit in Computer Networks. The project requires students to design, implement, and analyse, a client-server software architecture, using a "hidden" server, desktop computers, and handheld, mobile, wireless devices. Students were required to employ an existing pre-defined protocol to communicate with a black-box server, and to design their own protocol communicating between their own two software components. Burd et al. (2012) have also summarised this work.

While this project employed (jailbroken) Apple iPod devices running iOS, as that was matched by our department's resources and knowledge at the time, there are no specific iOS requirements. Setting a similar project in coming years would likely benefit from the use of Android instead of iOS, and enable more students to use their own mobile devices. Interested readers are warmly invited to contact the author for copies of the software, and for instructions suitable to getting students started.

References


Oblinger, D. & Oblinger, J. L., eds (2005), Educating the net generation, EDUCAUSE, Boulder, CO.


Teaching Model-Driven Software Development: Revealing the “Great Miracle” of Code Generation to Students

Andreas Schmidt\textsuperscript{1,2} \hspace{1cm} Daniel Kimmig\textsuperscript{2} \hspace{1cm} Klaus Bittner\textsuperscript{2} \hspace{1cm} Markus Dickerhof\textsuperscript{2}

\textsuperscript{1} Department of Informatics and Business Information Systems
Karlsruhe University of Applied Sciences
PO Box 2440, 76012 Karlsruhe, Germany
andreas.schmidt@hs-karlsruhe.de

\textsuperscript{2} Institute of Applied Computer Sciences
Karlsruhe Institute of Technology
PO Box 3640, 76021 Karlsruhe, Germany
\{daniel.kimmig | klaus.bittner | markus.dickerhof\}@kit.edu

Abstract
A didactic approach into teaching model-driven software development (MDSD) is proposed in this paper. The main idea is to focus on conveying underlying concepts, rather than managing a concrete tool or presenting a purely theoretical approach, when teaching MDSD. This objective shall be reached by the development of a simple code generator by the students. For this reason the whole process from graphical modeling to the actual code generation is traversed twice. The first time from back to front to introduce the main concepts of a code generator engine and in a second pass from the beginning to extend the generator by additional functionality. The course will then be completed by transferring the knowledge learnt to a concrete generator tool within the framework of a simple exercise and by a presentation.

Keywords: Model driven Software Development, Teaching, Practical Approach.

1 Introduction
Interest in model-driven software development (MDSD) has recently increased, due in part to the Model Driven Architecture (MDA) initiative of the Object Management Group (OMG). Consequently, the industry’s demand for graduates qualified in this field has grown. It is therefore considered crucial to educate students in MDSD concepts, tools and techniques.

1.1 Model Driven Software Development
The idea behind MDSD concerns designing a model of the system to be developed (for instance, using UML), which is then transformed into code in a specific target language, such as PHP, C# or Java. Using this technique, there is a strict distinction between the domain-specific and technological aspects of a system. While domain specific concepts are represented in the model (as classes, attributes and methods), technology specific aspects like the target language are determined in the transformation rules, specified by the programmer. Typically the mundane and repetitive parts of the code can be easily generated using this technique, so the programmer can focus on the more complex and challenging parts of a system. Benefits of this approach include higher quality and consistency of code, easier translation to newer or different technologies, as well as shorter time in development (Stahl, Voelter, and Czarnecki 2006).

2 Existing Approaches
In developing a didactic concept for our course, the approaches of other universities that teach in this field were examined. Additionally, the proceedings of the Educators’ Symposium (EduSymp; Seidl, M. and Clarke, P., 2010), a major forum for software modeling education, was used as a resource. EduSymp is collocated with the annual ACM/IEEE international conference on Model-Driven-Engineering Languages and Systems (MODELS). The results of studying and interpreting these sources yielded three possible variants of MDSD instruction:

1. \textit{Purely theoretical approach}: An advantage of this approach is that it is focused on the most relevant concepts of MDSD. However, this approach faces the drawback in that students often do not consider it to be engaging. This is the case at universities of applied sciences (UASs), which focus strongly on practical work. Nevertheless, a number of universities were found to be using this approach, although some were partly supported by practical lectures.

2. \textit{Tool-supported approach}: An advantage of this approach is that there is a good initial point for quick initial learning by the student. However, this approach is associated with the risk of the students learning how to use a specific tool, but neglecting to learn the concepts. A case study at EduSymp 2011 (Seidl and Clarke 2011) showed that the Eclipse Modeling Framework (EMF) is
the dominating tool in this field. EMF (Steinberg and Budinski 2009) is an extension of Eclipse. It allows the specification of a structured data model and the subsequent generation of Java classes from this model.

3. **Practical approach**: This program-driven approach focuses on conveying underlying concepts rather than the use of a concrete tool. This can be performed by the development of a simple code generator, based on tools that can be commonly used by students. The advantage of this approach is that it is more engaging for the students. One risk with this approach is the danger of excessive workload demand on the students.

3 Surrounding Conditions
The course is integrated in the undergraduate curriculum of a degree program in Business Information Systems at a UAS in Germany. In contrast to traditional universities, UASs have a more applied or practical orientation, and less research (Fachhochschule 2013). As a consequence, they have Bachelor and Masters programs, and not PhD programs. With their strong focus on teaching professional skills, the graduates are oriented for an industry position, rather than academic.

The course should occur at the end of the Bachelor program (6th semester) as a compulsory optional subject of two contact hours per week (2 credits). At this point, the students have already finished courses in Programming (I + II), Modeling, Software Engineering (I + II), Databases (I + II) and Operating Systems. Additionally they have a number of business economics and mathematic based courses. For a complete overview of all courses see (HSKA 2013).

During the last two academic terms of their studies, students can choose from a wide spectrum of different elective courses. The interest of the students for a special subject, as well as the student’s availability of time, plays a crucial role in the students’ decision to take a course.

Although these students have a solid base in programming skills, it is important to consider that the students are business informatics candidates, rather than computer scientists. Therefore, students have an abstract view of software generation before enrolling in this course, as depicted in Figure 1.

**Figure 1: Broad idea of a code-generator**

4 Our Approach
From the three possible variants of teaching MDSD, as previously articulated, the pure theoretical approach is eliminated. The reasons for this were as follows:

1. This approach often is not seen as engaging by students. Therefore, in order to compete with courses in subjects, such as mobile business or social media (which are very popular at this time), it is unlikely that this this approach is perceived as attractive by students.

2. Additionally, students at UASs have a stronger focus on practical work, therefore a theoretical approach would seem to be less appropriate in this learning environment with an applied focus.

Because the construction of a code generator could be a complex task, it was decided to use an existing code generator in our course. With this approach, students could get an initial feeling of success quite early in the course. For that reason, a number of different code generators were examined, but none were deemed satisfactory, the main reason being that many of the available generators are very complex or specialized, and not easy to use. Therefore, there is a danger that students will only learn to handle a particular tool, and the concepts of MDSD will have less emphasis. Due to its wide use in MDSD-related courses, the Eclipse Modeling Framework (EMF) was examined, yet rejected. The lack of dedicated tools for teaching MDSD was also one of the big challenges formulated in the position paper about Software Modeling Education from Seidl and Clarke (2011).

Parallel to the evaluation of the different code generator tools, the topics and central concepts of the course were selected. A number of core topics were identified, as well as a number of optional topics considered as “nice to teach”, depending on time constraints. The core topics comprise of:

- Code generator types
- Models
- Meta-models
- Model transformation
- Model verification
- Template systems
- XML Metadata Interchange (XMI)

Domain specific languages (DSL) are disregarded, due to time constraints (Fowler 2010).

Considering the slightly-exaggerated initial view of a code-generator held by students (See Figure 1, above), one of our main efforts would be to remove the confusing area of the “miracle cloud” in the middle. Using an existing code generator tool like EMF leads to the danger that this cloud will remain, because the “miracle” happens inside the tool. To address this, a third approach in teaching MDSD (the practical approach) was considered, which involves letting students build a generator by themselves. This should lead to students to understand the whole process of code generation. In this case, no “miracle cloud” will remain because students will gain insight as to how code is generated. This insight will aid students in their future use of various code generators, or construction of small or medium size generators for specific problems.

The development of the code generator is accomplished by a number of consecutive exercises. In order to reduce the complexity, the capability of the generator will be limited to the generation of a subset of UML class diagram capabilities (classes, attributes, and relations). Additionally, the initial meta-model will be provided in the form of an extensible library. In this way,
the complete workflow starting from graphical modeling to actual code generation is conveyed.

4.1 Tools and Technologies
A careful selection of appropriate tools to support the implementation of the code generator has to be done. Because there are only two contact hours a week with students, tools and languages are selected because they are largely known by the students, and also because an excessive learning expenditure is not required to learn the tool. For instance, the use of a descriptive language, such as the Object Constraint Language (OCL) for implementing constraints (Warmer and Kleppe 2003), is disclaimed. Instead, the constraints are formulated in a procedural manner. At this point, it will also be referred to the fact that other often more comfortable, but also more complex solutions exist. Understanding the generator principles however is not limited by these simplifications. It enables the student to concentrate on the most important points.

4.1.1 Programming Language
The code generator should be implemented using a scripting language, due to its generally high accessibility and prevalence, and good string handling. Possible candidates are Perl (Wall, Christiansen, and Schwartz 1996), Python (Lutz 2006), Ruby (Flanagan and Matsumoto 2008), and PHP (Lerdorf, Tatroe, MacIntyre, and Apani 2006). The language PHP was selected, due to the fact that our students have already used this language at this point in time, and that PHP is a so-called macro language, meaning that it already supplies a template mechanism.

4.1.2 Meta Model
To accelerate learning, a small PHP library is supplied to the students, which represents the initial meta model and allows the formulation of their models by an API. The meta-model allows the formulation of classes, attributes, and relations. In addition, the meta model supplies a number of methods or properties to iterate over the classes and their attributes. The meta-model represents a central component in the system to be implemented and is used or extended by the students at many points.

4.1.3 Template Engine
The template mechanism incorporated in PHP is used at the beginning of the course, but later on replaced by the explicitly available template engine Smarty (Hayder, Maia, and Gheorghe 2006), a widely-used module in the PHP community, which is considered to be highly mature. As well, Smarty has high quality documentation (Ohrt and Zmievski 2012).

4.1.4 UML-Modeling Tool
The only requirement made on the modeling tool is that an XMI export format can be written. This is provided by most modeling tools.

4.1.5 XML
The external model to be developed and the XMI model generated by the UML-modeling tool are both XML-based. For this reason, DOM, XSLT, XQuery and XPath functionalities are needed, which are provided from PHP by default. Xalan (Apache 2007) and Zorba (Zorba 2013) are used as external XSLT and XQuery transformation tools.

4.1.6 Workflow
Many steps are required to get from graphical modeling over various model transformations to the actual code generation. Often, certain parts of code generation (mostly transformations) are accomplished within the framework of development. Consequently, these parts shall not be delivered in a monolithic block, but exist as external components with clearly defined interfaces, which are then assembled by the UNIX Tool make (Oram and Talbott 1991).

4.2 Didactic Approach
Code generators are divided into three parts: generator-frontend (definition of model); generator-kernel (model transformation, verification, etc.); and generator-backend (code generation). A multi-stage approach was selected. In the first stage, the process is started at the back end (code generation), and worked successively to the front end. This way, the students see the result at the beginning (the source code generated) and can easily derive the necessity and capability of the upstream components. This derivation of requirements for upstream components is achieved by extending the initial task covering code generation. At the end of the first stage, a complete execution chain exists from modeling with an external UML modeling tool to several transformation steps, to the generation and formatting of the source code. The second stage is aimed at extending the code generator with additional capabilities, such as inheritance or supporting other UML diagram types, such as state transition diagrams. Contrary to the first stage, it proceeds in the other direction. In other words, the extensions are made starting from the output of the UML modeling tool to the meta model, to code generation by templates.

Figure 2 gives an overview of the exercises in the first two stages. Due to this double treatment of basic components of a code generator, in-depth understanding of the functioning of a generator is obtained, which exceeds the understanding obtained from learning a concrete tool or from the purely theoretical approach.
5 Exercises

A number of consecutive exercises are assigned to students of the course. Within the framework of these exercises and the accompanying theoretical lessons, the following concepts will be covered: templates, model, meta-model, model verification, transformation between the same as well as between different meta-models, XML-based models and meta-models, XMI.

5.1 First Stage - From Backend to Frontend

There are six exercises in the first stage. They are named 1) generating simple Java classes, 2) Supporting relationships, 3) use of the new Meta-model and explicit template engine, 4) model verification and transformation, 5) using an XML based model and 6) connecting a graphical UML modelling tool. These exercises are described in detail below.

5.1.1 First Exercise - Generating Simple Java Classes.

The starting point is the actual code generation. For this purpose, a simple Java source code consisting of some classes is presented to the students. Figure 3 shows a single class that should be generated. Then, the students are asked to implement a minimum PHP program that generates the given Java source code.

```php
class Person {
    protected String name;
    protected Date yearOfBirth;

    public Person() {} //empty constructor
    String getName() {
        return name;
    }
    void setName(String value) {
        name = value;
    }
    Date getYearOfBirth() {
        return yearOfBirth;
    }
    void setYearOfBirth(Date value) {
        yearOfBirth = value;
    }
}
```

Figure 3: Single Java class to be generated

It is now the task of the students to analyse the source code for parts that represent recurrent concepts for all classes and, hence, can be generalized and parts that are class-specific. In doing this, the students separate the model from the template part of a generator. Additionally, they implicitly define a simple meta-model for their model, consisting of an appropriate data structure. Lastly, they use the PHP macro feature as an example of a template system. Figure 4 shows a simplified result to be produced by the students.

5.1.2 Second Exercise - Supporting Relationships

In the next lesson, the model shall be extended to cover relationships (1:m, n:m) among classes. The first task the students have to perform is extending the syntax of the model. Most of the students’ solutions support predefined data types, as well as class names and lists, or a combination of both (i.e.

'yirector'=>>'Person:directs',
'actors'=>>'List(Person):plays').

The implementation of the semantics has to also be done. This is realized in the template part. Here, the templates tend to get a little bit cluttered. At this stage students generally realize that their model includes a number of weak points. Firstly, the model is completely unrestricted and secondly it is error-prone, in the case that wrong information is provided.

As an extension of this exercise, students are asked to generate DDL code for a relational database. At first sight this looks not more difficult than generating the Java code, but ends up in an unpleasant surprise. The fact that the relationships are implemented with foreign keys does not allow the generator to sequentially access the model information and transform the current element into the target code. Such an operation requires random access to the model schema to obtain the information about the corresponding primary key and its data type.

These problems will be used as motivation to work on an improved meta-model, which is then presented to the students. The meta-model allows the definition of the model via a number of API calls. The meta-model also allows accessing the information in the model by navigating along properties, and has an extension mechanism to meet special needs for the different target languages. A basic example is given in Figure 5, illustrating the use of the API. In the upper part of the figure, the definition of a model (Name: Demo) with two classes and a relationship between them is shown. In the lower part, the “how to iterate” over the model information is illustrated. In this example, the information about the properties and the primary key of each class is
printed. The corresponding ER-model is shown in Figure 6.

```java
// Definition of model:
//
$ord = new Model('Demo');
$ord = $ord->createClass('Order');
$ord = $ord->addAttribute('id', 'int');
$ord = $ord->addAttribute('title', 'String');
$ord = $ord->addAttribute('year', 'int');
$ord = $ord->addAttribute('score', 'int');

// Iteration over the attributes of a class:
//
foreach ($class->attributes as $att) {
    print "Att: $att->name, type: $att->type, name: $att->name\n";
}
```

Figure 5: Definition of the model from Figure 6, using the Metamodel-API

5.1.3 Third Exercise - Use of the new Metamodel and explicit template engine.

When generating the database schema in the last exercise, because of the required non-linear access to the model, the templates often get “contaminated” with snippets of PHP code, which made the template difficult to read and maintain. In order to address this topic, a simple solution is presented to our students. When utilizing an explicit template engine (Smarty), cluttering the templates with PHP code snippets is not possible. A template system has a small number of specialized language elements, which are well suited for generating arbitrary output, but not to solve general programming problems. Tasks for providing the required template data can be provided by extending the given meta-model or by extending the template system by user defined functions and so called ’modificators’. These techniques keep the templates small and readable. For that reason, in this exercise, students have to build the model with the new provided API-based meta-model and also to adapt their existing templates from the PHP-macro functionality to Smarty.

Additionally, students have to overcome the problem of platform-specific data types for Java and the DDL code. To facilitate the implementation of the templates for the different targets, students also extend the provided metamodel with appropriate methods, which are mostly specific to the target platform. Figure 8 shows a possible solution developed by the students. The methods is1N(), is11(), getCard1Side() and getCardNSide() are extensions of the provided metamodel implemented by students. Additionally, for solving the problem with the different datatypes in Java and MySQL, the students have to extend the functionality of the Smarty template engine with appropriate modifiers (i.e. mysql_type).

5.1.4 Fourth Exercise - Model Verification/Transformation.

In this exercise, a number of tasks have to be performed on the meta-model. Firstly, some verifications of the model must be implemented. Secondly, each class in the model is extended by some administrative fields (which demonstrates a model to model transformation on the same meta-model). Finally, a transformation to another, more platform-specific meta-model (with concepts such as table, foreign key, relationship table, etc) must be implemented. Figure 7 shows an example of a basic model transformation within the same meta-model, by adding two administrative fields to each class.

```java
foreach ($model->classes as $class) {
    $att = $class->addAttribute('creationTime', 'date');
    $att->addAttribute('modificationTime', 'date');
}
```

Figure 7: Simple Model to Model Transformation

5.1.5 Fifth Exercise - Using an XML based Model

In this exercise an XML-based language, which represents the concepts in the meta-model, has to be developed by the student. Next, an import filter that maps the developed language to the metamodel API must be implemented.

5.1.6 Sixth Exercise - Connecting a Graphical UML Modeling Tool

In the sixth and last exercise in the first stage, a graphical modeling tool is connected. This has a number of steps. First, the XMI language has to be analysed. Afterwards an XSLT or XQuery transformation from XMI to the previously developed XML language must be carried out. Figure 9 shows an example of a simple XML-language, covering the model concepts. With this step, the complete workflow from graphical modelling of the features to be generated, to a series of transformation and verification steps, to actual code generation has been implemented. In the subsequent extension step (second stage), the generator is extended with additional features in the opposite direction.
5.2 Second Stage - From Frontend to Backend

The exercise in the second stage consists of extending the functionality of the code generator by additional features, such as another UML diagram type (i.e. state transition diagram) or the extension of the class diagram by other features (inheritance, methods, additional attributing by stereotypes and tags). Extension of the generator consists of the following tasks.

First, the students have to analyse how the newly supported UML language elements are expressed in the XMI format. Then, they have to extend their own XML language by the additionally needed language elements.

Afterwards, the XSLT/XQuery style-sheets must be adapted to support this transformation as well. This extension then continues over the generator-internal meta-model, the import filter, and ends with the development of new templates to generate the additional code. Figure 10 shows the coverage of the individual exercises inside the whole workflow of the generation process. It can be clearly seen that this approach starts from the backend of the generator.

5.3 Third Stage - Implementing a Concrete Task with an Existing Generator Tool

The course is then completed by analysing an existing code generator tool, which is freely available and chosen by the student. The objective is to handle a simple code translation task with the tool, and to present this tool to other students in a presentation of about twenty minutes in length. In this presentation, all students should refer to the concepts presented in the course.

6 Evaluation of our Approach

The course has been offered once a year since 2006. After students present results of an existing code generator or related tool/technology at the end of the course, they are asked for their impression about the didactical concept. Interestingly, even the students who at the beginning of the course would have preferred the use of an existing tool (mostly, because they already have some experience in MDSD) rather than building a code generator itself, changed their mind and favor the chosen approach. The insight they received from building a small but complete code generator from scratch by far exceeds their prior understanding of such tools. Further, students who had no previous experience with MDSD stated that building the code generator helped them in understanding the chosen generator at the end of the course, as knowledge of the internals of a code generator was acquired at that time. Additionally, students feel prepared to be able to develop small or medium size code generators in the future. Even so, some students are intimidated when they find out that they have to build a generator in this course, prompting them to choose another course.

An empirical study from Whittle and Hutchinson (2011) investigated how industry uses MDSD produced findings relevant to the instruction of MDSD. First, a lot of MDSD examples concern small generators and DSLs developed in as little as two weeks. Another interesting finding of the study was that successful MDSD practice starts from the ground and doesn’t follow a heavyweight top-down approach. Top-down means that students have to develop abstract models first, refine them into architecture, and then finally to code. Formulating abstractions of a system before the details are full...
understood is quite a difficult job, compared to the bottom-up approach with starts for looking after reusable assets in the code and then creating abstractions from them. This can be compared with the approach described in this article, starting at the backend of a generator, leading the student to understand the necessity of components introduced as the course progresses. With a start at the beginning (XMI), there would have been much more problems to motivate further steps.

7 Summary and Outlook

The paper describes a didactical model to convey the fundamentals of MDSD in the classroom environment. In an approach consisting of several stages, the most important concepts in the field of MDSD are learned, detailed, and applied in practice. As of 2013, the course has been lectured 8 times. In the evaluations, students mention a higher workload than average, compared to other elective courses, but nevertheless rank it as one of their favorite courses. In particular, the first stage starting from the backend and motivating the need of further MDSD-specific concepts is highly appreciated, which can be documented by means of the very high rating for the central theme.

While the actual platform for implementing the code generator is PHP, Smarty and make, it is planned to also provide some exposure to the Java framework (using ant and freemarker) for the future. This framework should not replace the existing PHP solution but should be an alternative for students who prefer Java instead of PHP.

Another approach currently under development is providing students with a code generator skeleton which represents the state after the first stage as a starting point, and continues by extending the generator with additional functionality.

8 Acknowledgement

We like to thank Michael Dohan from Lakehead University in Thunder Bay, Ontario, Canada, for proofreading of paper, and translating it into scientific English – thanks Michael!

9 References


How difficult are novice code writing tasks? A software metrics approach

Jacqueline Whalley and Nadia Kasto
School of Computing and Mathematic Sciences
AUT University
Auckland, New Zealand
{jwhalley,nkasto}@aut.ac.nz

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

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

1 Introduction
There is a plethora of literature in computing education pointing to the fact that novice programmers find programming particularly difficult and that assessing the knowledge and skills the students have gained is problematic (for example see: Robins, Rountree & Rountree 2003). Historically the pass rates for students undertaking first year courses have been relatively low. This in part might be due to some difficulties related to the assessment of these courses. Whalley et al. (2006) noted that “assessing programming fairly and consistently is a complex and challenging task, for which programming educators lack clear frameworks and tools” (p. 251). More recently, Elliott Tew (2010) suggested that “the field of computing lacks valid and reliable assessment instruments for pedagogical or research purposes” (p.xiii) and Whalley et al. (2011) noted that there is a need for “more consistent and equitable designs, an improved learning experience for the novice and an overall increase in the quality of teaching and assessment of novice programmers” (p. 45).

2 Background
In order to design better novice programming assessments computer science educators have attempted to apply various educational taxonomies. The most commonly adopted taxonomies to date are Bloom’s (Bloom, 1956), the revised Bloom’s (Anderson et al., 2001) and the SOLO Taxonomy (Biggs & Collis, 1982). One of the strengths of the use of educational taxonomies such as Bloom’s and SOLO for guiding the design of assessment is that they are designed to consider the level of thinking and in the case of the revised Bloom’s taxonomy also the knowledge required in order to successfully solve a problem. However, the use and interpretation of Bloom and the revised Bloom’s taxonomy has proved to be problematic (for example see: Fuller et al. 2007, Thompson et al. 2008, and Shuhidan et al. 2009). In a recent, study Gluga et al. (2012) suggested that many of the ambiguities in the application of Bloom’s taxonomy to the assessment of computer programming are due to the necessity to have a deep understanding of the learning context in order to achieve an accurate classification. They also noted that the classifiers often had preconceived misunderstandings of the categories and differing views on the complexity of tasks and the sophistication of the cognitive processes required to solve them.

Researchers have reported that SOLO can be reliably used to classify code reading questions and the student responses to those questions as long as the classifiers have a shared understanding of the application of the taxonomy to code comprehension tasks (Clear et al. 2008, Sheard et al. 2008). An initial set of guidelines and descriptors for using SOLO to classify student code writing solutions were proposed by Lister et al. (2009). However, classifying student answers to code writing tasks using this interpretation of the SOLO levels proved difficult even with these guidelines (Lister et al. 2009, Shuhidan et al. 2009). A novel combination of SOLO and Bloom’s revised taxonomy was used by Meerbaum-Salant, Armoni and Ben-Ari (2010) to guide the design of assessments.

In a more recent study, Whalley et al. (2011) found that by combining a framework of salient elements and code quality factors they were able to more clearly define the SOLO categories. Using this approach they were able to reliably apply the principles of SOLO to determine the level of a code writing task or problem. However the programming tasks that they analysed were from various programming examinations and written using pen and paper rather than a computer.

The body of research into using SOLO for classifying questions and student responses to both comprehension and writing questions has consistently reported that the higher the SOLO level of a question the more difficult, as measured by student performance, the question was (Clear et al. 2008, Sheard et al. 2008, Whalley et al. 2011).

Although some progress has been made towards being able to classify and estimate the difficulty of code-comprehension questions “we have no reliable measure of the difficulty of code-writing questions even at the macro level” (Simon et al. 2009). While SOLO and
Bloom maybe useful, given the nature of novice code writing questions we really need reliable measures at a higher level of detail than taxonomies such as SOLO can provide in order to be able to research the nature of these questions and their role in assessment.

2.1 On metrics and instructional design
Starsinic (1998) used an interpretation of the English language Flesch-Kincaid readability measure (Kincaid et al. 1975) to produce a script (in Perl) called Fathom that was designed to automatically measure the readability of code generated by junior programmers during time critical projects.

In a later study Börstler, Caspersen and Nordström (2007) proposed that some cognitive aspects of code reading can be expressed using common software measures and explored this idea in the context of two novice code reading tasks. Their aim was to develop a reliable means of selecting appropriate code examples to help guide novice programmers’ learning and to determine between good and bad examples. They surmised that a good example must be readable and comprehensible and designed a framework based on these principles. Their framework consisted of cyclomatic complexity (McCabe 1976), an interpretation of the English language Flesch Readability Ease Measure (Flesch 1948) and Halstead’s difficulty metric (Halstead 1977).

In subsequent work a software metrics approach to informing the design of code comprehension assessments, for novice programmers, was reported by Kasto and Whalley (2013). This work adopted a goal oriented approach to the identification of software metrics for measuring the difficulty of code comprehension tasks. In this study the difficulty of a question was represented as the percentage of fully correct answers provided. Novel dynamic metrics were designed specifically to measure the complexity of code tracing tasks and were shown to correlate, along with cyclomatic complexity and average block depth, significantly with the difficulty of the task. They also investigated the use of metrics for explain in plain English (EiP) questions but did not find any significant correlations between difficulty and Halstead metrics or cyclomatic complexity but noted that this may have been an artefact of the assessment questions that were used in the study. The authors concluded that software metrics may be a useful tool to assist in the design and selection of questions when setting an examination and that code writing tasks might also be amenable to the same approach by identifying relevant software metrics and applying them to the model answer and to the student solutions.

In this paper we report on preliminary attempts to use software metrics as a way of estimating the difficulty of code writing tasks.

3 Software Metrics
“Good code is short, simple, and symmetrical – the challenge is figuring out how to get there”. –Sean Parent

There are no software metrics that measure code which has yet to be written. Because we are aiming to develop an objective means of measuring the difficulty of a novice code writing task prior to the students undertaking the task we elected to use the instructor’s model answer as the code for which the metrics are calculated. While the model answer might provide a better quality solution that solution might actually have less complex code than many of the answers elicited from the students. In order to write the better answer the students may have to produce a more generalised, connected or integrated solution that reduces redundancy (Whalley et al. 2011). The challenge for developing a metric, for measuring the difficulty of a code writing task designed for novice programmers, is finding a measure that measures the level of quality of the code not just the structure of the code. This view is supported by Börstler, Caspersen and Nordström (2007) who reported that measures, for code examples, that are suitable for use in an educational context must also take into account factors such as level of thinking required and cognitive load. This must also be the case for code writing tasks.

3.1 Code structure metrics
When writing code it is necessary to come up with a structure for the code. Regardless of the quality of the solution we expect that some code structure metrics should have some relationship to the relative difficulty of code writing tasks.

The software metrics that have been shown to correlate to code tracing task difficulty measure the structure of the code and/or the data flow of the code when executing. These metrics are:
- McCabe’s cyclomatic complexity (McCabe 1976),
- average nested block depth,

and two novel “dynamic metrics” for code tracing tasks (Kasto and Whalley, 2013):
- Sum of all operators in the executed statements
- Number of commands in the executed statements.

In tracing code only the paths of code that the students must trace though are adding to the complexity. In the case of code writing all paths are important so the dynamic metrics are not considered to be as relevant for code writing questions. As a consequence the metrics we selected for code writing were cyclomatic complexity and average nested block depth.

It is important to acknowledge the limitations of these metrics. Cyclomatic complexity in particular has been the subject of considerable criticism (for details see Shepperd 1988, Piwawaski 1982, and Magel 1981). Cyclomatic complexity “directly measures the number of linearly independent paths through a program’s source code” (McCabe 1976). However in calculating cyclomatic complexity statements such as else, do and try, object creation and method calls are not considered. It is highly likely that these statements contribute to the complexity of a code writing task for novice programmers. However given that cyclomatic complexity and average nested block depth were found to correlate with the difficulty of code comprehension tasks we elected to evaluate them again here for code writing tasks.

Driven by the limitations of common software complexity metrics, Magel (1981) proposed a complexity metric based on regular expressions. Magel represented the structure of a piece of code as a control flow graph
and then derived a regular expression from the control flow graph. The symbols in the regular expression were then counted to give the complexity structure metric. Full details of the calculation of this metric can be found in Magel’s paper. Magel surmised that “confusing program segments require longer regular expressions” and therefore a higher value for his metric (p.63). Because the quality of the model solution may prove to be more predictive of difficulty than the structure of the code we also selected Magel’s regular expression metric for evaluation. We hypothesised that questions that provide the opportunity for solutions that are more refined (more generalised, connected or integrated) have a higher regular expression metric and are likely to be more difficult questions than those that do not have the potential for refinement.

Additionally we used the following structural metrics:

- The total number of commands; the number of Java method calls.
- The total number of operators
- The number of unique operators

We included the number of commands because as an artefact of using a micro-world almost all of the procedures written required the students to call methods on objects. The number of commands metric measures the number of Java method calls in the model answer. Both number of operators and number of unique operators were included because we were interested to know whether it is the total number or the number of different operators required that increases the difficulty.

3.2 Code readability metrics

A basic prerequisite for understandability is readability (Börstle, Caspersen and Nordström 2007). In order for code to be readable the basic syntactical elements must be easy to recognize. Only then, can relationships between the elements be established which may then lead to an understanding. It is reasonable to include a metric that measures the readability of code (i.e. of the model answer for a novice programming question) because empirical research has found that there is a strong relationship between the ability to explain code and write code with pen and paper (Lopez et al., 2008).

Readability metrics have been developed and applied to natural languages. These language measures generally produce a single numeric value, which indicates either the grade level (1-12) or readability (usually 1-100) of a document and which is constructed from the average number of syllables per word and the average number of words per sentence.

Although these natural language metrics are far from perfect, and despite their apparent simplicity, they have been found to be useful in practice. One of the most commonly used measures, the Flesch-Kincaid metric (Flesh 1948) is integrated into popular text editors and has been in used for over 50 years. However, these measures don’t map well onto code therefore simply running a prose-readability test on student code would not generate a useful measure (Starsinic 1998).

The Software Readability Ease Score (SRES) is an adaptation of the Flesch Reading Ease Score where the lexemes of the programming language are interpreted as syllables, its statements as words, and its units of abstraction as sentences (Börstle, Caspersen and Nordström 2007). This metric was designed on the premise that the smaller the average word length and the average sentence length, the easier it is to recognize relevant chunks (units of understanding). Unfortunately the authors did not provide the detail for the calculation of the metric.

Starsinic (1998) developed a similar metric where he opted to measure the number of tokens per expression (e.g. +, ;, {, &, and any keyword), the number of expressions (e.g. 0.2 and (5a + 6)) per statement (e.g. a = $foo::bar * 7;)) and the number of statements per Perl subroutine. His final formula was:

\[
\text{code complexity} = \frac{(\text{average expression length in tokens}) \times 0.55}{\text{average statement length in expressions}} \times 0.28 + \frac{(\text{average subroutine length in statements}) \times 0.08}{\text{average expression length in tokens}}
\]

The paper concluded that a low Starsinic readability metric value indicates a more readable piece of code and that a piece of code with a readability of 2.91 was very readable whereas code with a readability of 6.85 was considered to be very complex and therefore hard to read.

No justification or explanation is provided for the weightings given to each operand in the formula or for the thresholds that were used to determine the relative level of complexity of the code readability.

We elected to start from Starsinic’s readability metric but we altered the way in which expressions are counted. For example, in Starsinic’s method an expression such as \(n = n + 1\); would count as one expression but we counted this as two expressions in an attempt to more closely map the way in which a novice might read the expression. We think it is likely that a novice would break this down into two expressions firstly evaluating \(n + 1\); and then evaluating the assignment.

4 Dataset

The eleven code writing questions analysed in this study were selected from a series of controlled, summative practical programming tests held throughout the first semester of a first year Java programming course. The course adopts a back to basics procedural approach (similar to that suggested Reges (2006)) except that the learning is supported by an in-house micro-world called Robot World in the BlueJ IDE. For each question the students were provided starting code with unit tests, as a BlueJ project, and asked to add a method to that project (see Appendix A for the questions). Sixty student responses were analysed for each question. These students had given ethical consent for their data to be used and were representative of the entire cohort.

5 Analysis

Table 1 gives the software metrics and student performance for each of the questions analysed. It should be noted in interpreting the analysis that difficulty is being measured as the percentage of fully correct answers. For example question 11 is the easiest question with a percentage difficulty of 100% whereas question 1 was the most difficult question with 14% of students giving a correct working solution.
Cyclomatic complexity, average nested block depth, number of operators and number of unique operators were calculated using the standard procedures provided by the Rationale® Software Analyzer 7.1 (RSA 2013) tool. The regular expression metric and the readability metric were calculated by hand.

The significance of the correlation of each metric to the difficulty of each question was then tested using a Pearson’s correlation (Table 2).

Cyclomatic complexity, the regular expression metric and the readability metric were found to all correlate strongly with the difficulty of the novice code writing questions that we analysed in this study.

The higher the cyclomatic complexity, the more complex the control flow of the program code is and the more difficult the question is (as evidenced by a low percentage of students getting the answer correct).

The more deeply nested the branches of the code are the higher the average nested block depth is and the more difficult the question was for the students. This is not really surprising. Research investigating student responses to code writing questions found that students find questions that can be solved by writing the code line by line with limited reference to the previous lines of code are easier than those that require the students to understand the relationship between the chunks or blocks of code that they have written (Whalley et al. 2011).

Similarly we found that the higher the number of Java commands required the more difficult the question is.

For the regular expression metric a higher value results from nested code (Figure 1, A vs. B), backward branches rather than forward branches (Figure 1, D vs. C) and increasing complexity in selection statements (Figure 1, E and F).

The strong correlation between difficulty of the question and increasing structural and data flow complexity, as measured by the regular expression metric, confirms our original hypothesis and supports the conjecture that many students cannot write code that requires more complex structures and that there must be some relationship between the ability to design code structure and being able to produce working code regardless of the quality of their code.

Given that we are analysing the instructor’s model answer, we are assuming that it is good code. If there are, for example, nested blocks to reach this solution a relatively high level of integration of the code and merging of plans is likely to be required. For such a question there are usually several solutions that could provide a working solution. If the student’s solutions are of a lower quality than the instructor’s code then you could argue that the code produced by them is more confusing and that the students would find it hard to correct any errors in their code. This could make the question more difficult for the novice programmer than the analysis of the model answer would indicate. The readability measure also correlates strongly to difficulty. The easier the model answer code is to read the easier the code is to write. It is possible that there is a causal relationship between readability of code and the ease of writing.

### Limitations

While the findings of this study are encouraging there are some caveats.

We have only examined a relatively small set of code writing questions. The questions were selected to cover the key topics taught in our first year programming course; sequence, selection and iteration. The sequencing of the questions within the tests could add to our observed difficulty of the question. However with such strong correlations it is unlikely that this effect would significantly alter our findings.
The questions we have analysed are also limited to “unseen” questions presented to student in a test situation. If previously seen questions are included it is likely correlations with the metrics used here will be less significant or even not significant. The difficulty of the question would be affected by the level of thinking required. A problem for which the students have already seen the code may mean that students can simply answer the question by recall.

Much of the reasoning around why we are seeing the relationships between the metrics and actual difficulty is based on conjecture and this aspect of the work could be improved by observing the students in the tests.

Some of the metrics used in this study may not be generalizable to all teaching contexts or indeed to all novice programming tasks. Courses that adopt an objects first pedagogy may have writing tasks for which other object orientated metrics might be applicable such as cohesion and coupling metrics. For a back to basics, algorithm focused, java course that does not utilise micro worlds but instead uses a typical IDE metrics such as number of commands may not be relevant. It is worth noting that for most metrics the range of values in a typical novice code writing task is likely to be relatively small. For example average nested block depth where deep nesting may be discouraged, by the instructor, in favour of separation of inner blocks into method calls. Despite the relatively small range of values we have found the metrics still correlate strongly with difficulty.

In selecting the metrics to use we believe that average nested block depth, cyclomatic complexity, regular expression metric and readability should provide a measure of difficulty of the task regardless of teaching approach and programming context. However other metrics would need to be selected based on the teaching approach. Some aspects of the teaching approach will be reflected in the model answer. For example, if considering a typical programming task such as printing a box of asters of any size the model answer may be a solution that has two for loops while another instructor’s model answer may consist of a nested loop.

While you could argue that as experienced teachers we consider these aspects of a programming problem when setting an assessment it is still useful to have a method for objective evaluation of the difficulty of a code writing question prior to including it in an assessment.

7 Future work
Where to from here?
Further analysis could be undertaken to examine which metrics are general predictors of difficulty of novice programming tasks. Moreover metrics could be identified that are useful for specific pedagogies.

If we can establish suitable heuristics for selection of metrics for a given course it may be possible to use this approach to automatically grade code writing tasks. We may even be able to use metrics as a tool for providing immediate feedback to the students about the quality of their solutions. Good code must be simple, readable and comprehensible and we want our students to be producing quality code. However in this study, we do not consider the quality of the students solutions in determining the difficulty of a question – a fully correct
answer may not be a well-designed answer. If you were wishing to adopt metrics to assist in the grading of student work then perhaps some measure of distance of the student’s answer from the instructor’s model answer might be useful. Some work has been undertaken which investigates the usefulness of software metrics as a form of formative feedback for novice programmers (Cardell-Oliver 2011). This work used program size metrics, unit tests and program style violation counts as forms of automated feedback. While software metrics such as the ones we have explored in this paper are difficult for novice programmers to interpret directly, if supplied with guidelines for interpretation it is possible that students might also find them a form of useful feedback.

Finally we believe that this approach has value as a research tool and provides a way of comparing questions in an empirical manner. However, we concur with Börstle, Caspersen and Nordström (2007) that measures that are suitable for use in an educational context must also take into account factors such as level of thinking required, cognitive load and instructional design.

Metrics should not be used as a silver bullet but used in conjunction with more subjective measures of difficulty such as SOLO or Bloom’s classification which consider the level of thinking and/or knowledge required.

8 References


Appendix A: The questions

1. This question asks the students to write a method that makes the robot clean the room. The robot must pick up all the beepers left lying around and if there are enough beepers to fully load the beeper wash then they should be loaded into the beeper washer (at location (2, 12)) any remaining beepers should be neatly placed at location (2, 0). The students are supplied with the method signature and unit tests to test that the beepers have been dropped at the appropriate location(s). The tests include starting worlds with 0, 5, 9, 10, 15 and 20 beepers.

2. This question asks the students to write a method called advanceRobot that has two parameters a Robot and a distance to travel (the number of cells that the robot should advance). The robot should only be able to move if it is alive and if the distance to travel is positive if it is unable to move an appropriate exception should be thrown. If the robot encounters a wall before moving the full distance it should stop rather than crashing. The method should return true only if the robot moved the full distance.

3. In this question the students must write code to move the robot from a set starting location at (4, 0) to a fixed exit at location (4, 6). In order to do this the robot must choose one of two paths. If there is a beeper at the first intersection (4, 2) then the robot must follow the eastern path otherwise it should follow the western path.

4. In this scenario there are two corridors with a gap between them. The length of each of the corridors changes randomly every time the World is created, but the gap is always in the same location.

5. The students were provided with a method header and asked to write a summing algorithm; write code that makes a robot move forwards until it reaches a wall while picking up any beepers that it encounters and then print out the total number of beepers the robot collected.

6. Complete the method findBeeper that moves the robot through a spiral maze until it reaches a beeper. You should also count how many steps the Robot navigate to the beepers and return the number of steps required.

7. A robot starts in one of two possible initial states, as shown in the figures below:

Write a program to move the robot to the end of the corridor. If the robot starts at location (0, 0), it must finish at location (4, 4) facing north. If the robot starts at location (0, 5), it must finish at location (4, 1) facing south.
<table>
<thead>
<tr>
<th></th>
<th>In this question the students are provided with a robot in a cell that contains a number of beepers. The students are asked to write a method called <code>pickUpNBeepersCheckIfAll()</code> that takes an integer parameter, and makes the most recently created robot pick up that number of beepers from the beeper stack at its current location. You can assume that there are enough beepers in the stack for the robot to do this safely. The method should return true if the robot has picked up all the beepers at its current location, or false if there are still beepers on the ground.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Write a method called <code>pickUpBeeperStack()</code> that makes the most recently created robot pick up all the beepers at its current location. The method should return no value and take no parameters.</td>
</tr>
<tr>
<td>10</td>
<td>For this question the students are supplied with the method header they are asked to complete the method body so that the robot turns left then if there is no wall in the way moves forward one cell.</td>
</tr>
<tr>
<td>11</td>
<td>For this question the students are supplied with the method header they are asked to complete the method body by writing a sequence of three statements to make the robot drop the beeper it is carrying, then move the robot forward one cell and turn the robot left once.</td>
</tr>
</tbody>
</table>
Benchmarking a set of exam questions for introductory programming

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Julian Dermoudy
University of Tasmania
Australia
julian.dermoudy@utas.edu.au

Daryl D’Souza
RMIT University
Australia
daryl.dsouza@rmit.edu.au

Minjie Hu
University of Otago
New Zealand
minjie.hu@otago.ac.nz

Dale Parsons
Otago Polytechnic
New Zealand
dale.parsons@op.ac.nz

Abstract
This paper reports on the combining of two related but hitherto distinct themes in programming education research. The first is the recognition that students in programming courses tend to perform far more poorly than their teachers would like, and further, more poorly than their teachers would expect without a careful analysis of their results. The second is the proposal of a number of different styles of examination question, sometimes coupled with analysis of student performance on those questions, typically at single institutions. This work combines these themes by including a common set of short questions in the final examinations of introductory programming courses at six institutions in Australia and New Zealand, and analysing the student performance across all six institutions. The analysis results in a set of four simple questions that can be used to benchmark student performance in introductory programming courses at a wide range of institutions.

Keywords: Standards, quality, examination papers, CS1, introductory programming, assessment.

1 Introduction
While researchers have long observed that students have difficulty learning to program (e.g. Pea (1986); Soloway et al (1982)), the McCracken study (2001) brought broader attention to the issue by establishing that the problem is not constrained to individual courses in individual institutions but is widespread, and can come as a surprise to the academics teaching the courses in question.

An extreme interpretation of the work of the McCracken group is that many programming students cannot program. Yet it is clear that many students pass programming courses, so a number of researchers and projects have subsequently set out to examine more closely the assessment in programming courses, to try to establish how the assessment of programming aligns with the skills and knowledge that students are expected to acquire. Researchers working in this area include Simon et al (2010), Petersen et al (2011), and Sheard et al (2013).

In recent years, moreover, increasing attention has been paid to standards in higher education, with government bodies such as Australia’s Tertiary Education Quality and Standards Agency (2012) charged with evaluating the performance of higher education providers against a new Higher Education Standards Framework. The combined discipline group for Information and Communication Technology (ICT) and Engineering has begun its quest for learning standards by drawing on existing learning outcomes developed from the relevant professional bodies (Cameron & Hadgraft, 2010).

The work reported here forms part of the BABELnot project (Lister et al, 2012), a principal goal of which is to explore a possible approach for the development and assessment of learning standards in programming courses. Formal written examinations are a common form of assessment in programming courses, and typically the form to which a large portion of marks are attached. In the work reported in this paper we have developed and tested programming assessment questions across multiple institutions with the aim of establishing a set of questions that can be used as a benchmark against which student performance can be measured.

2 Benchmarking
Benchmarking is the process of evaluating performance against an established standard. Benchmarking of student performance is often undertaken in universities and the results are used in a number of ways at the course, department, and institution level.
Benchmarking of student performance is an important tool for quality assurance. It may be used to establish minimum or acceptable levels of performance in a course or to compare course performance across cohorts or institutions (Woolf et al., 1999). The results of benchmarking may be used for marketing purposes or to make strategic decisions at a department or institution level.

Benchmarking may also be used to make improvements to a course or program of study. It can be used in evaluation of teaching approaches, resource provision and student selection. An example of the use of benchmarking in the computing education discipline is Oliver’s (2000) work on developing benchmarks for IT literacy.

There are other benefits of benchmarking exercises. The development of a benchmark requires a community effort to examine and determine appropriate criteria and standards. The communication and collaboration required serves to strengthen the community. Sim, Easterbrook and Holt (2003) propose that “benchmarking, when embraced by a community, has a strong positive effect on the scientific maturity of a discipline” (p.74).

2.1 Benchmarking programming performance

There is a large corpus of research on the learning and teaching of introductory programming. Many studies have attempted to determine factors which influence learning outcomes. Although there are many positive findings, most studies are conducted in the context of a single course, making it infeasible to conduct comparisons across different institutions or different research studies. An analysis of 164 papers reporting research into the learning and teaching of programming found that 72% of the studies were conducted within one institution and most of these were within a single course (Sheard et al., 2009).

Several studies have investigated student performance in programming across multiple and international institutions. These include work by McCracken et al. (2001), Lister et al. (2004) and the BRACElet project (Clear et al., 2009/2010; Whalley et al., 2006). However, these studies have focused more on investigating performance of students to understand what and how they learn rather than the development of suitable questions which could be applied more widely for future benchmarking exercises.

The BABELnot project (Lister et al., 2012) has proposed a number of possible examination questions, along with a scheme for classifying questions according to a number of distinct criteria (Sheard et al., 2013). Harland et al. (2013) examined the performance of students in two programming courses, with the aim of determining how the criterion of degree of difficulty might be measured for particular questions. The attempt was to establish a calibration for the expectations of instructors about the difficulty levels, but also as a means of examining what it means for a question to be considered difficult. On the basis of BABELnot-designated exam questions in just two exams they concluded that the absolute scale of low, medium and high is not appropriate for classifying questions (or at least not for classifying questions based on students’ results). By contrast, however, Simon et al. (2012) established that researchers trained in the classification system can accurately assess the difficulty of examination questions on this same three-point scale.

The work in this area has been hampered by a scarcity of standard test questions that can be used to measure performance. One widely used example is Soloway’s ‘rainfall problem’ (Soloway, 1986). Since the development of this question in the early 1980’s, it has been used in a number of comparative studies of student performance. However, it has recently been suggested that this question is not suitable for current use (Simon, 2013).

The aim of our benchmarking study was to develop a set of questions that could be used by introductory programming educators at multiple institutions as a standard to measure the performance of their students.

3 Research approach

The work presented here was conducted as a follow-up to a workshop held in conjunction with the Australasian Computing Education conference in 2013 (ACE2013). The “Writing a good exam for a programming course” workshop was held as part of the BABELnot project (Lister et al., 2012). The aim of the workshop was to explore different styles of questions used for assessment of introductory programming students. A planned outcome was a set of introductory programming exam questions suitable for use in a benchmarking exercise across multiple institutions.

In preparation for the workshop, members of the BABELnot project team compiled a list of questions that were considered possible candidates for the benchmarking exercise. A total of 76 questions were sourced from exam papers from five institutions. Five project members then individually assessed the suitability of each question according to the criteria:

1. Is the question likely to be used by others?
2. Is there a clear marking guide? (to help achieve consistency in marking for benchmarking)

Based on the results of this assessment the list was trimmed to 34 questions.

Prior to the workshop the registrants were sent the set of 34 questions and asked to assess their suitability for use in an introductory programming exam. For each question they were asked to choose one of three options:

1. I would like to use this question
2. I would consider using this question
3. I would not use this question

Twelve people responded to the survey. Their responses were collated and the questions sorted according to their popularity.

Seventeen people participated in the workshop. All had taught or were currently teaching an introductory programming course. During the workshop the results of the survey were presented to participants. The idea was to provoke discussion about what makes a good or poor programming exam question and what would make a question unsuitable for use across multiple institutions. First, the least popular questions were presented and participants discussed potential issues with these
questions. Next, the most popular questions were presented and participants gave reasons why they would use these questions and any possible issues with their use. Finally, the remainder of the questions were discussed. This middle group were the most controversial as there were no clear decisions as to their suitability.

There were a variety of reasons why questions were deemed unsuitable for use in a multi-institutional benchmarking exercise. Issues identified:

- 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.

When all the questions had been discussed, participants were asked to reconsider the questions and vote for questions that they would be prepared to use in a benchmarking study. From the results of the voting a set of 11 questions were chosen. The final selection aimed for a spread across topics, question styles, skills required to answer the question and the type of response required from the student. A brief description of the questions is shown in Table 1.

At the end of the workshop a benchmarking study using the 11 questions was proposed, and six participants from four Australian and two New Zealand institutions agreed to continue with the study.

### 3.1 Benchmarking study

The benchmarking study was conducted during first semester in 2013. Each person obtained ethics approval from their institution and arranged for the questions agreed upon at the workshop to be placed in their final exam. The order of the questions and the marks allocated to each question were decided by each participant. However, to enable valid comparison of results the questions were kept in the same style. An exception to this was at one institution, where two questions were converted from short-answer to multiple-choice format. The responses for these questions from this institution have been omitted from the analysis.

The questions were originally designed in Java and were converted to C# or Python as required. This required minimal changes to the presentation of the questions. The greatest change required is that with one multiple-choice question concerning iteration, the obvious off-by-one error in the Java implementation was one less than the actual number of iterations; but when this was translated to Python, the obvious off-by-one error was one more than the actual number of iterations. This was accounted for by a change in one of the incorrect multiple-choice options, and a change in the order of the options.

At the end of semester the students’ responses and marks gained for each question were collated for analysis.

#### 3.1.1 Course profiles

A profile of the programming courses used in the study is shown in Table 2. All courses taught introductory programming, with five at the undergraduate level and one at the postgraduate level. The latter course is effectively the same as courses taught to first-year undergraduate students, but is taught to students who are taking a postgraduate computing qualification to supplement a degree in some unrelated area.

The language of instruction, IDE, and programing paradigm approach varied across the six courses in the study. However, all covered the basic programming constructs (expressions, assignment, sequence, selection, iteration and arrays) that are covered in the questions used in the study.

#### 3.1.2 Exam profiles

A profile of the exams from which the data was collected for the benchmarking study is shown in Table 3. All exams were written and held at the end of the course. The exams comprised from 30% to 50% of the total assessment marks in their respective courses. The questions are worth 110% of the exam for course 5 because the exam consisted of just these questions, one of which was marked as a bonus question.

### Table 1: Questions for benchmarking study

<table>
<thead>
<tr>
<th>Question</th>
<th>Style</th>
<th>Skill required</th>
<th>Open/closed response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Assignment &amp; Sequence</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection A</td>
<td>Short response</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection B</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection C</td>
<td>Short response</td>
<td>Explain code</td>
<td>Open</td>
</tr>
<tr>
<td>Iteration A</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration B</td>
<td>MCQ</td>
<td>Explain code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays A</td>
<td>Short response</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays B</td>
<td>MCQ</td>
<td>Explain code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays C</td>
<td>Short response</td>
<td>Write code</td>
<td>Open</td>
</tr>
<tr>
<td>Iteration and Arrays D</td>
<td>Code writing</td>
<td>Modify code</td>
<td>Open</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>Style</th>
<th>Skill required</th>
<th>Open/closed response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Assignment &amp; Sequence</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection A</td>
<td>Short response</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection B</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Selection C</td>
<td>Short response</td>
<td>Explain code</td>
<td>Open</td>
</tr>
<tr>
<td>Iteration A</td>
<td>MCQ</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration B</td>
<td>MCQ</td>
<td>Explain code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays A</td>
<td>Short response</td>
<td>Trace code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays B</td>
<td>MCQ</td>
<td>Explain code</td>
<td>Closed</td>
</tr>
<tr>
<td>Iteration and Arrays C</td>
<td>Short response</td>
<td>Write code</td>
<td>Open</td>
</tr>
<tr>
<td>Iteration and Arrays D</td>
<td>Code writing</td>
<td>Modify code</td>
<td>Open</td>
</tr>
</tbody>
</table>
Table 2: Course profiles, with asterisks indicating the four courses that were used in the final benchmarking

<table>
<thead>
<tr>
<th>Course</th>
<th>Country</th>
<th>Level</th>
<th>Programming language</th>
<th>IDE</th>
<th>Teaching approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*</td>
<td>NZ</td>
<td>U/G</td>
<td>C#</td>
<td>Visual Studio 2010</td>
<td>Procedural</td>
</tr>
<tr>
<td>2*</td>
<td>Aus</td>
<td>U/G</td>
<td>Java</td>
<td>Eclipse</td>
<td>Objects later</td>
</tr>
<tr>
<td>3</td>
<td>Aus</td>
<td>P/G</td>
<td>Java</td>
<td>BlueJ</td>
<td>Objects first</td>
</tr>
<tr>
<td>4*</td>
<td>Aus</td>
<td>U/G</td>
<td>Java</td>
<td>JCreator</td>
<td>Objects early</td>
</tr>
<tr>
<td>5</td>
<td>NZ</td>
<td>U/G</td>
<td>C#</td>
<td>Visual Studio 2012</td>
<td>Objects later</td>
</tr>
<tr>
<td>6*</td>
<td>Aus</td>
<td>U/G</td>
<td>Python</td>
<td>JES</td>
<td>Objects later</td>
</tr>
</tbody>
</table>

Table 3: Exam profiles, with asterisks indicating the four courses that were used in the final benchmarking

<table>
<thead>
<tr>
<th>Id</th>
<th>Open/closed book</th>
<th>Questions used</th>
<th>% of exam</th>
<th>Exam % of course</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*</td>
<td>closed</td>
<td>11</td>
<td>47</td>
<td>35</td>
</tr>
<tr>
<td>2*</td>
<td>closed</td>
<td>11</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>closed</td>
<td>11</td>
<td>16</td>
<td>50</td>
</tr>
<tr>
<td>4*</td>
<td>open</td>
<td>10</td>
<td>36</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>open</td>
<td>11</td>
<td>110</td>
<td>30</td>
</tr>
<tr>
<td>6*</td>
<td>closed with 'cheat sheet'</td>
<td>11</td>
<td>35</td>
<td>50</td>
</tr>
</tbody>
</table>

4 Results

This section presents the results of our benchmarking study. Data of student performance on the set of 11 questions was collected from six institutions. The number of exam papers from each institution varied from 13 to 297, as shown in Table 4. The first step in the benchmarking exercise was to establish which institutions to include in the analysis. Our aim was to find questions upon which students over multiple institutions showed similar performance. For the benchmarks to be useful it is important that they can be used by other institutions and so we wanted to compare student results across the courses and exams with similar profiles. To assist in this process we conducted a preliminary inspection of the student performance results. The graph in Figure 1 of the percentage of correct responses to the closed questions (questions for which there was only one correct response and therefore no interpretation needed by the marker) shows that the results from two institutions appear to be outliers. The postgraduate group results (course 3) were higher than the other results for all but one question. Further inspection showed that the marks for each open question were also higher than for the other questions. Including the postgraduate cohort would make it less likely that we would find questions with similar results. It was therefore decided not to include this group. At the lower end of the graph, the results for course 5 were lower for most questions and substantially lower for several. This course has a very small enrolment, and is usually assessed solely by practical assessments. The exam was included expressly for this research, and the students were not accustomed to written exams. In the circumstances, it seemed appropriate to omit this course, too, from the analysis.

The benchmarking analysis, conducted to find questions on which there was no significant difference across comparable course, was therefore conducted on the remaining four undergraduate courses. The remainder of this section presents the questions themselves and our findings on the comparability of the students’ performance of the students in courses 1, 2, 4, and 6.

4.1 Expressions

This question tests students’ knowledge of a boolean expression with relational and boolean operators.

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>(a) age &lt; 18 &amp;&amp; salary &lt; 10000</td>
<td></td>
</tr>
<tr>
<td>(b) age &lt; 18</td>
<td></td>
</tr>
<tr>
<td>(c) age &lt;= 18 &amp;&amp; salary &lt;= 10000</td>
<td></td>
</tr>
<tr>
<td>(d) age &lt;= 18</td>
<td></td>
</tr>
</tbody>
</table>

Overall, 85% of the students selected the correct response (a). Almost half of the remainder (7%) selected (c), suggesting an error with evaluating the relational operators. Fewer students (3%) selected (b), indicating less difficulty with the logical operator. By course, the percentage of correct responses varied from 78-89%. The proportions of correct responses were compared using a chi-square test and no difference was found at $p < 0.05$. This marks this question as one that can be used for benchmarking across multiple institutions.

Table 4: Exam papers from each institution

<table>
<thead>
<tr>
<th>Course</th>
<th>Number of papers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*</td>
<td>36</td>
</tr>
<tr>
<td>2*</td>
<td>238</td>
</tr>
<tr>
<td>3</td>
<td>76</td>
</tr>
<tr>
<td>4*</td>
<td>297</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6*</td>
<td>166</td>
</tr>
</tbody>
</table>
4.2 Assignment and sequence

This question tests students’ knowledge of assignment and sequence and their ability to trace simple code.

Assignment & Sequence
What are the values of girls, boys, and children after the following code has been executed?

```java
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

Most students (83%) selected the correct response (c). Most of the remainder (11%) selected (d), suggesting either that they evaluated the assignment operator but not the sequence of operations, or that they consider the assignment in the fourth statement to be some kind of statement of continuous state rather than a single operation in a sequence (Simon, 2011).

By course, the percentage of correct responses varied from 75-86%. The proportions of correct responses were compared using a chi-square test and no difference was found at p < 0.05.

4.3 Selection

This question requires students to trace a nested if statement.

Selection A
This question refers to the following code, where the variables p, q, and result all have integer values:

```java
int p = 1;
int q = 2;
int result = 4;
if (p < q) {
    if (q > 4) {
        result = 5;
    } else {
        result = 6;
    }
}
```

What would be the value in the variable result after the code is executed?

Overall, 94% of the U/G students calculated the correct value of 6. The next most frequent answer was 5 (3%), indicating a problem evaluating the second if statement and then 4 (2%), indicating a problem evaluating the first if statement.

By course, the percentage of correct answers varied from 86-94%. A chi-square test showed this difference was significant, which means that the question cannot be reasonably used for multi-institutional benchmarking.

\[ \chi^2 (3,737) = 10.035, \ p < 0.05 \]

It has subsequently been suggested that not only are p and q uninformative names (which is intended), they are poor variable names for code in an exam question because they will be easily confused by students with dyslexia.

The next question also requires students to trace a nested if statement.
Consider the following block of code, where the variables $a$, $b$, and $c$ each store integer values:

```csharp
if (a > b) {
    if (b > c) {
        answer = c;
    } else {
        answer = b;
    }
} else if (a < c) {
    answer = c;
} else {
    answer = a;
}
Console.WriteLine(answer);
```

In relation to the above block of code, which of the following values for the variables will cause the value in variable $b$ to be printed?

(a) $a=1; b=2; c=3$
(b) $a=1; b=3; c=2$
(c) $a=2; b=1; c=3$
(d) $a=3; b=2; c=1$

Overall, 82% of the students selected the correct response (c). The next most frequent response was (d) (9%), indicating a problem evaluating the second if statement.

By course, the percentage of correct answers varied from 72-82%. The proportions of correct responses were compared using a chi-square test and no difference was found at $p < 0.05$.

The final selection question is a code-explaining question asking students to determine the purpose or outcome of the same piece of code that was used in the preceding question.

In one sentence, describe the purpose of the above code (ie 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.

Overall, 36% of the students gave a fully correct answer and 43% a partially correct answer. The remaining 21% gave answers that were awarded no marks.

The mean mark varied from 52-70%. A Kruskal-Wallis test showed these differences were significant.

$\chi^2 (3,704) = 41.213, p < 0.05$

### 4.4 Iteration

The first iteration question checks whether students understand a simple `while` loop. The loop iterates too many times to be easily traced, although a student who is unsure would be able to substitute a smaller number, trace the code, and then deduce the answer for the larger number.

```
count = 0;
while (count < 357)
{
    balance = balance + deposit;
    count = count + 1;
}
```

How many times will the body of the following loop be executed?

(a) 1
(b) 356
(c) 357
(d) 358

Overall, 68% of the U/G students selected the correct response (c). The next most frequent response was (b) (21%), and then (d), (8%) indicating one-off errors in calculating the number of iterations. The low number of responses for (a) (4%) indicated that most students understood that the number of iterations was determined by the loop condition.

The percentage of correct answers for the U/G group varied from 50-76%. A chi-square test showed this difference was significant.

$\chi^2 (3,736) = 21.172, p < 0.05$

The next iteration question, a multiple-choice code-explaining question (Simon & Snowdon, 2011), asks students to determine the purpose of a simple `for` loop.

```
int result = 0;
for (int i=1; i<=value; i++)
{
    result = result + i;
}
```

What is the purpose or outcome of the following piece of code?

(a) to add a counter to a result
(b) to count the numbers from 1 to `value`
(c) to add all the numbers from 1 to `value` – 1
(d) to add all the numbers from 1 to `value`

Overall, 51% of the students selected the correct response (d). The next most frequent response was (b) (21%). While this is clearly a nonsensical explanation (the answer would be the same as `value`), this is potentially what we earlier called a tricky question, as some students might misread `i` as 1 in the assignment statement within the loop body. A different name should be used for the loop counter in future versions of this question. The combined response for (c) and (d) indicated that 63% of the students considered that the purpose was adding all the numbers.

By course, the percentage of correct answers varied from 39-64%. A chi-square test showed that this difference was significant.

$\chi^2 (3,737) = 23.495, p < 0.05$
4.5 Iteration and arrays

This code-tracing question requires students to trace a while loop operating on two arrays of integers.

**Iteration & Arrays A**

What will be the value of result after the following code statements are executed?

```java
int[] val1 = {1,-5,2,0,4,2,-3};
int[] val2 = {1,-5,2,4,4,2,7};
int result = 0;
int i = 0;
while (i < val1.Length)
{
    if (val1[i] != val2[i])
    {
        result = result + 1;
    }
    i = i + 1;
}
```

Overall, 64% of the students calculated the correct answer (2). The next most frequent response was 1 (13%), and then 5 (9%).

The percentage of correct answers varied from 42-72%. A chi-square test showed this difference was significant.

\[ \chi^2(3,737) = 45.834, p < 0.05 \]

Another multiple-choice code-explaining question (Simon & Snowdon, 2011) asks students to determine the purpose of a for loop containing a selection statement that processes the elements of an array.

**Iteration & Arrays B**

What is the purpose or outcome of the following piece of code?

```java
int result = 0;
for (int i=0; i<=nums.Length; i++)
{
    if (nums[i] < 0)
    {
        result = result + 1;
    }
}
```

(a) to find the smallest element in the array of numbers
(b) to count the negative numbers in the array of numbers
(c) to count the numbers in the array of numbers
(d) to sort the array of numbers

Overall, 82% of the students selected the correct response (b). The next most frequent response was (c) (10%). The combined responses to (b) and (c) suggest that most students understood that the loop involved counting.

The percentage of correct answers varied from 77-85%. The proportions of correct responses were compared using a chi-square test and no difference was found at \( p < 0.05 \).

The first of the two code-writing questions in the set asks students to write some code (about three lines) comparing the elements of an array with their indexes.

**Iteration & Arrays C**

Suppose you had an array of integers called myArray.

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.

Overall, 56% of the students gave a fully correct answer and 24% a partially correct answer. The remaining 20% were awarded no marks.

The mean mark varied from 62-75%. A Kruskal-Wallis test showed that these differences were significant.

\[ \chi^2(3,710) = 22.776, p < 0.05 \]

The second code-writing question (shown in Figure 2) gives students a block of code that performs a certain operation on an array, and asks them to write code that reverses that operation. It is worth noting that a number of the participants at the workshop felt that this question was too hard; that one participant of the study declined to use the question because of its difficulty; and that one other participant included it as a bonus question, making it possible for students to score more than 100% on the exam. Overall, 32% of the students gave a fully correct answer and 48% a partially correct answer. The remaining 20% were awarded no marks.

The mean mark for the U/G group varied from 47-66%. A Kruskal-Wallis test showed that these differences were significant.

\[ \chi^2(2,679) = 39.213, p < 0.05 \]

4.6 Summary of results

Four of the eleven questions showed no significant difference between performance results over the four institutions for our benchmarking exercise. These questions were all multiple-choice; three were code-tracing questions and one was a code-explaining question.

The overall percentage of correct responses and the lower and upper range values across institutions are shown in Table 5.

The questions together cover six introductory programming topics. We propose that these questions could be used to benchmark undergraduate student performance in introductory programming courses that teach using Java, C# or Python; that introduce objects from an early to late stage; and that assess with open-book or closed-book exams.

<table>
<thead>
<tr>
<th>Question</th>
<th>Number of papers</th>
<th>% correct responses</th>
<th>Lower and upper range across institutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>737</td>
<td>85</td>
<td>78-89</td>
</tr>
<tr>
<td>Assignment &amp; Sequence</td>
<td>737</td>
<td>83</td>
<td>75-86</td>
</tr>
<tr>
<td>Selection B</td>
<td>736</td>
<td>82</td>
<td>72-82</td>
</tr>
<tr>
<td>Iteration &amp; Arrays B</td>
<td>737</td>
<td>82</td>
<td>77-85</td>
</tr>
</tbody>
</table>

Table 5: Suitable benchmarking questions
5 Discussion

The workshop at ACE provided an opportunity for 17 programming educators from 13 institutions in Australia and New Zealand to discuss and decide upon questions for the benchmarking exercise. The robust debate and ultimate consensus on the final selection of questions gave a firm foundation for this study. The involvement of educators from six different institutions in two countries should give wide acceptability to the findings of the study, with useful outcomes that may be more widely used by programming educators and applied in future research studies. Further benefits to this work include the illuminating discussion about exam questions, which was useful for reflection on assessment practices. An exercise of this type, with a practical outcome, also helps to strengthen the computing education community.

While we began this study with student performance data from six institutions, it quickly became clear that students at two of the institutions had performed quite differently from those at the other four. As we were seeking questions on which performance was comparable at a range of institutions, we excluded the two non-comparable institutions from our analysis. The four remaining institutions still represent a broad range: two of them are metropolitan universities in Australia, one is a large regional university in Australia, and one is a polytechnic in New Zealand.

Having found four questions on which there was no significant difference in performance across the four institutions, we proposed these questions as a benchmark. We were then effectively able to apply that benchmark to all six institutions, concluding with some circularity that the four institutions meet the benchmark, while one of the others fails to meet it and one exceeds it.

6 Conclusions and future work

The results of our study can serve as a benchmarking tool to compare learning outcomes of introductory programming students across courses, institutions and countries. Based on our findings we wish to extend this set to cover other core introductory programming topics.

It must be made very clear that we are not suggesting that the four questions identified here are all that needs to be assessed in an introductory programming course. What we are suggesting is that inclusion of these questions in the final exam of an introductory programming course will permit the assessor to form an idea of how the students in that course compare with the students at the three Australian and one New Zealand institutions analysed in this benchmarking exercise.

We are also not suggesting that these questions should set the standard of difficulty in the final exams of programming courses. All four of the exams in the benchmarking exercise included questions that were substantially longer and more difficult than those used here. Furthermore, our study did not consider possible differences in ability levels of students. All we are saying is that when eleven common questions were included in the final exams in those four courses, on four of those questions there was no significant difference in the performance of the students; and therefore that it should be possible to use those questions as a benchmark in other courses.

Future work will include an expansion of the question set to cover additional topics, and, if further participants are forthcoming, confirmation of these findings at a larger number of institutions.

7 Acknowledgements

The authors thank the Australian Government’s Office for Learning and Teaching (OLT) for their support of the ACE 2013 workshop and of the BABELnot project.

---

**Iteration & Arrays D**

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.

![Diagram](image)

Here is the code that performs that shift for an array called `values`:

```csharp
int oldRight = values[values.Length - 1];
for (int i = values.Length - 1; i > 0; i--)
    values[i] = values[i - 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.

---

**Figure 2: the final iteration and arrays question**
8 References


Simon (2013): Soloway’s rainfall problem has become harder. In proceedings of the First International Conference on Learning and Teaching in Computing and Engineering (LaTiCE 2013), Macau.


Developing A Framework to Assess Students’ Contributions during Wiki Construction

Iwan Handoyo Putro1, Angela Carbone2, Judy Sheard3

1Faculty of Information Technology
Monash University Australia
ian.putro@monash.edu

2Office of the Pro Vice-Chancellor (Learning & Teaching)
Monash University Australia
angela.carbone@monash.edu

3Faculty of Information Technology
Monash University Australia
judy.sheard@monash.edu

Abstract

The emergence of Web 2.0 technologies and their increasing use in higher education have provided opportunities for building collaborative learning environments for students. Collaborative experiences are particularly beneficial for preparing students for their future workplace environments. Moreover, the creation and sharing of resources and information as afforded by Web 2.0 technologies can also improve a student’s learning experience.

Web 2.0 technologies such as wikis can be used to enable and support new and varied forms of group work learning. Wikis provide tools for measuring students’ contributions, such as the number, size and regularity of their contributions. However, the value of the students’ collaboration and interactions as they worked together as a group is more difficult to determine. This suggests a need for an assessment framework to evaluate the value of students’ contributions and their interactions in wiki-based group work assignments.

The framework was built from a review of the literature, drawing on relevant research for assessing group work. Future studies will trial this assessment framework in a real unit setting that applies wikis for group work.

Keywords: assessment, wikis, group work

1 Introduction

The emerging research area of Web 2.0 technology adoption in higher education for group work is in part reflecting the changing employment needs in industries. People work collaboratively on projects, whether they are within the same organization or other institutions, using online collaborative and communication tools, such as wikis (Kille, 2006).

With the increasing adoption of online collaborative applications in the workplace (Venables and Tan, 2009), it is essential that students should have an orientation towards this globally connected environment and are prepared to operate with it effectively.

Most universities deploy network infrastructures that provide online access for staff and students (Matthews and Schrum, 2003). The existence of network infrastructure together with the availability of various online collaborative-based Web 2.0 applications, such as wikis, enables universities to provide authentic online collaborative work experiences for their students.

The adoption of wikis to support teaching and learning in higher education has been the subject of extensive and prominent research from many different perspectives. Studies on wiki implementations in higher education include investigations of wikis to support teaching and learning (Cole, 2009), understanding the learner’s acceptance of wikis in their learning process (Hartshorne and Ajjan, 2009), examining how wikis help learners complete their tasks (Larussone and Alterman, 2009) and exploring how wikis can be used to build learning communities (Wheeler et al., 2008).

Although collaborative learning mimics the real world workplace environment, there are issues with setup and adoption for students (Cajander et al., 2012) and often students are reluctant to work in a group (Caple and Bogle, 2013). One of the main reasons why students do not enthusiastically involve themselves in group work is related to the competitive nature of student learning (Smith et al., 2011). Students are commonly assessed on an individual basis and consider other students as their competitors. Students’ entry to higher education is based on their individual academic performance. Such competition, however, can compromise effective group work functionality, in particular when students are asked to work with others in a group.

Recently, educational institutions have adopted wikis to support and evaluate group-based assignments (Waycott et al., 2010). Wikis enable educators to assess student’s contributions by capturing and recording their contributions in the wiki’s log file. Individual contributions captured in log files are usually assessed through measurable indicators such as counting the number of contributions, calculating the amount of time students engaged with the tool and determining how regularly students participated during collaborative tasks. Studies that focused on evaluating the value of students’ contributions and interactions between members of a group in online tasks were rarely found.
In this paper we present an assessment framework to evaluate wiki-based group work assignments. By combining quantitative analysis of students’ individual contributions, together with qualitative analysis of group work interactions, this assessment framework provides a comprehensive set of rules to classify the value of student’s individual contributions and their interactions.

We begin by discussing theory and related literature on group work and online group work assessment. The next section describes the development of our proposed assessment framework. Finally we propose future work and present our conclusion.

2 Research Background

2.1 Group Work

Jacques (1984), in his seminal study on Learning in Groups, defines a group as a collection of individuals that have some key attributes, such as: collective perception, needs, shared aims, interdependence, social organization, interaction, cohesiveness and membership. Another definition by Davies (2009) also points to shared aims and collaborative behaviours as key indicators for healthy group work performance.

Often the term team is used interchangeably with group. However, while groups and teams shared some common attributes, teams are usually created for specific purposes while groups could be formed spontaneously and not for a specific purpose (Davies, 2009). For example, in a sporting situation, a sports team is rightly called a team rather than a group because it has a specific goal to win its events and in the longer term to be the most successful club. However, in an educational context, the term group is commonly used.

Typically, a group of students work together for one semester on a specifically designed group task. Therefore, in this paper the term group will be used.

Group work requires collaboration and cooperation (Mahenthiran and Rouse, 2000), each member of the group must interact, build understanding, present and challenge the ideas of other group members. As a result, group work often requires students to work on an authentic activity that could be useful for a student’s later employability.

A group does not start off fully formed and functioning when it is initially created. In order to successfully work as a group, every member should recognise the stages of group work development. Tuckman’s (1965) seminal study proposed the team stages model to address that challenge. The model consists of four stages, namely: forming, storming, norming and performing. In the first stage, team members meet for the first time to introduce and share their commitment. In the second stage, everyone begins to see themselves as part of the team and start to challenge each other. At the third stage, the team manages to have one goal and come to a mutual plan. Finally, at the performing stage, the team has reached a good performance through people working effectively together. A decade later, a fifth stage called adjourning was added to accommodate task completion and breaking up of the team (Tuckman and Jensen, 1977).

The benefits of group work in higher education have been recognised amongst educators. Students benefit from deeper learning because they have to learn and share knowledge to produce a collaborative product (Entwistle and Waterston, 1988). Group work also provides a more active learning environment as students share knowledge and experience (Ruel and Bastiaans, 2003). The capability to work as a group is an essential skill that recruiters and employers often look for (Maguire and Edmondson, 2001).

Although there are many benefits to be gained from group work, there are also issues that can occur. The motivation of group members has been noted to be one of the most serious problems in-group work (Hutter and Diehl, 2011). Some group members may be reluctant participants in assessment tasks and be uncommitted to the aims of the group (and the course for that matter). Motivational issues can arise as a result. Examples of motivational issues associated with group work are social loafing and free riding. These issues have received considerable attention in the literature (Hall and Buzwell, 2013, Jones, 2013). Social loafing occurs when capable students reduce their effort in a project while producing a good performance when working individually. The cause of social loafing is free riding (Mulvey and Klein, 1998), a situation where some of the members of the group might enjoy the benefits of a group mark without giving adequate contributions.

2.2 Assessment of Group Work

The assessment of online collaborative learning involves evaluating a student’s individual contribution and their final group product. A student’s individual contributions in group work include: adding text, images, links, changing layout, sharing ideas, allocating and managing tasks and integrating members’ contributions. While the final group product of collaborative work can be found in the form of a software manual, book and case study report. The range of diversity of students’ contributions and their final group product make assessment for online collaborative learning activities more difficult to design compared with individual assignment tasks (Swan et al., 2006).

Assessment of group work should consider the balance of students’ individual contributions and final group product (Trentin, 2009).

Fair assessment practice plays an important role in-group work assessment. Group results could be lower if one or more members do not adequately contribute. Students are often unenthusiastic about working in a group assignment, as they do not want to be graded based on other students’ performances (Orr, 2010, Johns-Boast, 2010).

We propose that a comprehensive assessment of collaborative group work would involve three main considerations:

• Students’ individual contributions
• Group interactions
• Final group product

as shown in Figure 1.
Monitoring a student’s individual contributions can be achieved easily online by counting the number of their activities during group process (Macdonald, 2003). The drawback with this approach is excessive workload for academics. Research has been done by Farrell et al. (2013) to reduce this amount of workload by developing an online assessment tool called the Task Contribution System. This tool was designed to provide an evaluation system that enables an individual’s contributions being assessed within a group task. However, merely counting contributions or activities cannot be used as an effective indicator of the quality of the contribution (Schrire, 2004). Therefore we argue that academics must also consider the value of a student’s contributions.

On the other hand, Figure 3 depicts collaborative work. It is achieved when group members contribute both to their own work and other member’s work by interactively working as a group.

Judging the overall quality of the group product is the third component of group work assessment. This is normally achieved by measuring whether the final group product is complete (i.e. has met the task specifications) and is cohesive. Cohesiveness is an important consideration when evaluating group work, as it gives a measure of how well the group has worked together to produce a product in which the components are clearly connected and well.

To complete a group task, interactions occur between the group members. That form of interaction can be described as cooperative or collaborative work. Cooperative works happens when group members complete their own tasks, but rarely interact and work on each others task (Arnold et al., 2012). See Figure 2.

A recent study by Calvani et al. (2010) used a Moodle plugin to visualize forum discussion interactions enabling monitoring of group collaborative work. This plugin is useful for educators to reduce paper work, allowing them to assess students’ contributions by automatically classifying and tabulating student’s activity into pre-defined categories. Although it is considered useful to encourage students to interact and discuss their ideas, the lack of facility to collaboratively produce the final group product is the main drawback of forum discussion boards. Therefore, evaluation of group work in forum discussions can only assess the idea development and not the actual final group product.

**Figure 1: Three aspects of group work assessment**

To complete a group task, interactions occur between the group members. That form of interaction can be described as cooperative or collaborative work. Cooperative works happens when group members complete their own tasks, but rarely interact and work on each others task (Arnold et al., 2012). See Figure 2.

**Figure 2: Group working cooperatively**

**Figure 3: Group working collaboratively**

### 2.3 Online Group Work Assessment

Forum discussion boards were early online tools which were used in assessment of groups tasks. Forum discussion boards have been used to support collaborative learning since late 1990’s (Meyer, 2010). As a result, there is an extensive literature that examines how forum discussions can be implemented to increase students’ interactions (Rovai and Barnum, 2007), promote collaborative learning (Curtis and Lawson, 2001), assessed using a set of rubrics (Rovai, 2007) and evaluated through grading criteria to promote awareness during knowledge building process (Sorensen and Takle, 2002)

A recent study by Calvani et al. (2010) used a Moodle plugin to visualize forum discussion interactions enabling monitoring of group collaborative work. This plugin is useful for educators to reduce paper work, allowing them to assess students’ contributions by automatically classifying and tabulating student’s activity into pre-defined categories. Although it is considered useful to encourage students to interact and discuss their ideas, the lack of facility to collaboratively produce the final group product is the main drawback of forum discussion boards. Therefore, evaluation of group work in forum discussions can only assess the idea development and not the actual final group product.
2.3.1 Assessment of Group Work in Wikis
Web 2.0 technologies provide teachers with a new approach to engage students to work in a group. By integrating this technology into instruction, classrooms move from teacher-dominated environments to student-centred environments (Keengwe, 2007). Whether it is participating in a class discussion or a forum discussion, the technologies available to students in a Web 2.0 classroom increases the amount they participate (Brodahl et al., 2011). By allowing students to use various forms of Web 2.0 technology for their collaborative tasks, teachers give students an opportunity to learn for themselves and share that learning with their peers.

A wiki is one of a suite of Web 2.0 applications that have been widely adopted in higher education (Ebner et al., 2008). A wiki is a browser-based software tool which enables users to collaboratively write, modify and delete content from a web browser using a simplified mark-up language or a rich text editor (Larussone and Alterman, 2009). A wiki also provides a history facility to keep track of the modifications made by different users and to enable changes to be reversed if necessary. Wiki pages can be created and edited using simple text editing facilities that are provided as part of the wiki software. The original philosophy of the wiki was one of complete openness, with any web user able to modify the content. However, a wiki can also be set up so that only certain users can modify the pages by giving different access to particular users (O'Leary, 2008).

In an educational context, wikis can offer many benefits: they allow students to work together in a shared environment, with the progress of the work visible to all students, and to the teacher, at any time (Richardson, 2010). This visibility and sense of creativity and progress can be highly motivating (Trentin, 2009). Students can provide feedback on each other’s work, and help to improve it (Lundin, 2008). Wikis also allow for web documents to be structured and organized in different ways, and to be updated regularly. They therefore provide a valuable way for groups of students, and their teachers, to collaboratively develop and maintain learning resources.

Although wikis have a lot of advantages, some students find wikis rather formal environments, and miss the interactive and community aspects of a forum discussion (Hemmi et al., 2009). Study by Cajander et al. (2009) reveals that there was a lack of structure in wikis, as a consequence students should read almost every part of the page to find recently added information. Furthermore, Vratulis and Dobson (2008) discovered that students might not all be able to play an equal role in making contributions to a wiki. Some students dominate and others fail to participate fully, which means that the final group product may not be representative of all students’ perspectives.

A wiki is a suitable platform for this research because it is a naturally collaborative working tool in which log files record and track users’ contributions so that academics can monitor and evaluate each student’s contributions and activities. However, as far as the assessing the quality of students’ contributions and group interactions are concerned, wikis provide limited support. For instance in Mediawiki, several extensions and plugins were available to support group evaluation, however, most of them are either based on counting measurable activities or cosmetics related (e.g: monitoring and visualization) (Kubincova et al., 2012).

3 Framework Development
With the emergence and adoption of wikis to support online collaborative tasks, a new way of assessment has opened up. By analysing log files that capture online activities there is an opportunity to gain an insight into understanding the value of students’ contributions and interactions.

A theoretical framework will be developed to evaluate three aspects of wiki-based group task:

- Students’ individual contributions
- Group interactions
- Final group product

3.1 Individual Contributions
The basic method of assessing a student’s individual contributions in a wiki-based group work assignment is by measuring the student’s quantitative activities, such as: how many contributions they have made, and the size of their contributions (Trentin, 2009).

However, we argued that additional information should be added to gain deeper understanding on the value of student’s individual contribution. Together with the basic method to evaluate student’s contribution in wiki, we propose four additional attributes:

- Number of contributions
- Size of contributions
- Types of contributions
- Purposes of contributions
- Regularity of contributions
- Relevance of contributions

3.1.1 Types of Contributions
Commonly occurring contributions in wiki-based tasks are adding text, images or links as well as editing, deleting and moving contents.

Several studies to identify students’ contributions in wikis have been performed (Pfeil et al., 2006, Ehmann et al., 2008, Arazy et al., 2010). These studies identify the types of student’s contributions such as: add, delete, proofread, improve navigation and add link to references.

The study by Calvani et. al (2010) on group work interaction proposes a set of thinking types. These include: explain, connect, ask, edit, organize, suggest, revise and summarize. These thinking types were used to label each segment of a student’s conversation in a forum discussion. By utilizing these pre-defined labels for each conversation segment, the type of students’ thinking could then be mapped. Judd et al. (2010) also provided a similar method to categorise students’ contribution based on content analysis.

There are a lot of thinking types defined in Calvani’s et. al (2010) study as the nature of forum
discussion board is more of a *conversational style* (e.g. propose idea, explain, argue) rather than *writing style* used in a wiki (e.g. add, delete, edit). We draw on these thinking types, but simplify them for our purpose of classifying type of a student’s action while they are writing their contribution in wikis. Our proposed types of contributions are:

- **Add**: add one complete sentence or more.
- **Edit**: add, delete or move one word or more (but not a complete sentence), typo and grammar correction.
- **Delete**: delete a sentence or more.
- **Move**: move a sentence or more to other section of the text.

We outline the types of student contributions and its description in Table 1. For any activities which involve students working on their own contribution, we put label 0 after categories (e.g. A0, E0) while for the activities on other member’s contribution we use label 1 (e.g. A1, E1).

<table>
<thead>
<tr>
<th>Types of Contributions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>Add one sentence or more to their own work</td>
</tr>
<tr>
<td>A1</td>
<td>Add one sentence or more to other member work</td>
</tr>
<tr>
<td>E0</td>
<td>Add, delete or move one word or more (but not a complete sentence), typo correction, grammar, punctuation mark of their own work</td>
</tr>
<tr>
<td>E1</td>
<td>Add, delete or move one word or more (but not a complete sentence), typo correction, grammar, punctuation mark to other member work</td>
</tr>
<tr>
<td>D0</td>
<td>Delete a sentence or more (i.e. : paragraph) of their own work</td>
</tr>
<tr>
<td>D1</td>
<td>Delete a sentence or more (i.e. : paragraph) of other member’s work</td>
</tr>
<tr>
<td>M0</td>
<td>Move a sentence or more (i.e. : paragraph) of their own work to other section.</td>
</tr>
<tr>
<td>M1</td>
<td>Move a sentence or more (i.e. : paragraph) of other member work to other section.</td>
</tr>
</tbody>
</table>

**Table 1: Type of students’ contributions**

3.1.2 **Purposes of Contributions**

The second attribute of a student’s individual contributions is its purpose. This attribute is used to categorise a member’s participation based on the characteristics of their contributions. Meyer (2010) comments that there is a lack of communication features in wikis that prevents discussion amongst group members. A study by Tuckman and Jensen (1977) reveals that groups form and build understanding by knowing each other, managing and reorganizing their work. This is an indication that during wiki construction there is more than just content produced.

Therefore, we propose three labels to identify students’ contributions based on its purposes:

- **Content related** (e.g. adding text, images, links)
- **Social** (e.g. greeting, asking questions)
- **Organizational** (e.g. task distribution and due date reminder).

By labelling students’ contribution by the purpose of their activities, we can gain insights to each group members’ role (e.g. content builder, proof-reader).

3.1.3 **Regularity of Contributions**

Regularity is a measure of the distribution of a group member’s contribution over time. Regularity can be used as an indicator of collaborative behaviour. Calvani et al. (2010) in their research show that regularity can be considered as sign of individual responses to group needs.

Regularity could also be used to identify student’s motivation. Hutter and Diehl (2011) argue that evenly distributed contributions over time indicates a high motivation to complete the group task.

3.1.4 **Relevance of Contributions**

In a group assignment, students are required to contribute to the completion of a task. It is important that the contribution is relevant to the task and it has an appropriate level of quality. Together these give a measure of the value of the contributions. A valuable contribution should enrich existing work not just adding the length of the text.

3.1.5 **Summary**

Table 2 shows the summary of attributes of a student’s individual contribution.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of contributions</td>
<td>A count of contributions made.</td>
</tr>
<tr>
<td>Size of contributions</td>
<td>A measure of the size of the contribution (number of characters).</td>
</tr>
<tr>
<td>Type of contributions</td>
<td>Categories of contribution (e.g. add, edit, delete, move)</td>
</tr>
<tr>
<td>Purposes of contributions</td>
<td>Characteristics of contribution (e.g. content related, social, organizational)</td>
</tr>
<tr>
<td>Regularity of contributions</td>
<td>Distribution of contribution over time</td>
</tr>
<tr>
<td>Relevance of contributions</td>
<td>Extent to which the contribution improves the richness of current work</td>
</tr>
</tbody>
</table>

**Table 2: Individual contributions**

3.2 **Group Interactions**

The second aspect of wiki-based group work evaluation is group interaction and behaviour. In this aspect, there will be three attributes used for evaluation:
• Balance participation
• Cooperative interaction
• Collaborative interaction

3.2.1 Balance Participation
Developing a wiki as a collaborative process could end up as an unbalanced set of group activities (Calvani et al., 2010). This is because not all of the members will equally contribute during the group process. Some of the members will dominate the group by contributing large amounts of content while others will participate less.

In a healthy group, all members should participate equally without monopolizing or not contributing. Therefore, the attribute we introduce here is balance participation. It can be derived from both the contribution types and the purpose of contributions.

3.2.2 Cooperative Interaction
One of main drawbacks in wikis is the reluctance to contribute to other group member’s work. Students report that shared editing takes more effort rather than writing their own task (Ma and Yuen, 2008). Minocha and Thomas (2007) found that contributions from other students are not always accepted as constructive feedback. Wheeler et al. (2008) indicate that group members were resistant to having their contributions changed by their peers. While Twu (2010) suggests that cultural background plays an important role in this behaviour.

A study by Valente et al. (2012) reveals that the nature of activities influence interactional behaviour. If the task is not designed to promote cooperative behaviours, group members could work independently to complete their task. This type of group interaction can be identified as cooperative interaction. This type of interaction can indicate when group members are mostly concerned with their own task.

3.2.3 Collaborative Interaction
Collaborative behaviours have greater potential to improve the final product (Arnold et al., 2012). The more interactions happened in the group, the more chance for the group to reshape the content for a better, more cohesive product.

Consequently, if students are reluctant to work on each other’s work then the final product may appear to lack cohesion. In collaborative work, it is vital that members show reciprocal trust and sense of belonging in the group.

Collaborative behaviours can be identified by assessing whether the members’ contributions enrich other members’ work. Recent studies by Li (2012) and Mitchell and Carbone (2011) found that assignment specification should be carefully designed to promote collaborative learning.

3.2.4 Summary
A summary of the proposed attributes for assessing group interactions is shown in Table 3.

3.3 Final Group Product
The evaluation of a collaborative task can in part, be assessed by judging the completeness of the final group product (Macdonald, 2003). This type of assessment checks whether the task meets all the assignment requirements. However, evaluating the functionality of the final group product only illuminates the completeness aspect of a collaborative work. It can not determine how valuable students’ interactions were on the wiki during construction of the final product. Completing a wiki-based assignment, as a collaborative activity, is made up from lots of individual contributions from group members. As a result, a good final group product should show cohesiveness (integration and synthesis) from several individual group members.

Therefore, in addition to completeness, we proposed cohesiveness of contributions as an attribute to evaluate the final group product. Table 4 shows the summary of the propose attributes.

Table 3: Group interactions/behaviours

3. The Assessment Framework
We have described three aspects of assessment of wiki-based group work. Table 5 shows a complete picture of the proposed assessment framework that includes all aspects of assessment, attributes for each aspect.

There are three aspects of evaluation proposed: a student’s individual contribution, group interactions and the final group product. We have argued that the first aspect, student’s individual contribution, group interactions and the final group product. We have argued that the second aspect, group interactions, consists of three attributes: balance participation (level of balanced/equal participation across all members), cooperative interaction (working on their own task) and collaborative interaction

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance of participation</td>
<td>Level of balanced/equal participation in the interactions</td>
</tr>
<tr>
<td>Cooperative interaction</td>
<td>Amount of the time group members work on their own task</td>
</tr>
<tr>
<td>Collaborative interaction</td>
<td>Amount of the time group members take responsibility on other members’ task</td>
</tr>
</tbody>
</table>

Table 4: Final product attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completeness</td>
<td>Degree to which the final group product meets task specification</td>
</tr>
<tr>
<td>Cohesiveness</td>
<td>Degree to which individual contributions’ have been integrated and synthesised</td>
</tr>
</tbody>
</table>

Table 5: The Assessment Framework

4 The Assessment Framework
We have described three aspects of assessment of wiki-based group work. Table 5 shows a complete picture of the proposed assessment framework that includes all aspects of assessment, attributes for each aspect.

There are three aspects of evaluation proposed: a student’s individual contribution, group interactions and the final group product. We have argued that the first aspect, student’s individual contribution, group interactions and the final group product. We have argued that the second aspect, group interactions, consists of three attributes: balance participation (level of balanced/equal participation across all members), cooperative interaction (working on their own task) and collaborative interaction

A summary of the proposed attributes for assessing group interactions is shown in Table 3.
Table 5: Summary of proposed assessment framework to evaluate wiki-based group work construction

<table>
<thead>
<tr>
<th>Aspects of Assessment</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Contributions</td>
<td>Number of contributions</td>
<td>A count of contributions made.</td>
</tr>
<tr>
<td></td>
<td>Size of contributions</td>
<td>A measure of the size of the contribution (number of characters)</td>
</tr>
<tr>
<td></td>
<td>Types of contributions</td>
<td>Category of contributions (e.g. add, edit, delete, move).</td>
</tr>
<tr>
<td></td>
<td>Purposes of contributions</td>
<td>Characteristic of contributions (e.g. content related, social, organizational).</td>
</tr>
<tr>
<td></td>
<td>Regularity of contributions</td>
<td>Distribution of contributions over time.</td>
</tr>
<tr>
<td></td>
<td>Relevance of contributions</td>
<td>Extent to which the contribution improves the richness of current work.</td>
</tr>
<tr>
<td>Group Interaction</td>
<td>Balance of participation</td>
<td>Level of balanced/equal participation in the interactions.</td>
</tr>
<tr>
<td></td>
<td>Cooperative interaction</td>
<td>Amount of the time group members’ work on their own task.</td>
</tr>
<tr>
<td></td>
<td>Collaborative interaction</td>
<td>Amount of the time group members take responsibility on other members’ task.</td>
</tr>
<tr>
<td>Final Product</td>
<td>Completeness</td>
<td>Degree to which the final group product meet task specification.</td>
</tr>
<tr>
<td></td>
<td>Cohesiveness</td>
<td>Degree to which individual contributions’ have been integrated and synthesised.</td>
</tr>
</tbody>
</table>

(Contributing to another member’s task). The last aspect, the final group product has two attributes: completeness (meets project specifications) and cohesiveness (level of integration and synthesis).

5 Conclusion and Future Work
This paper describes the development process of an assessment framework, which can be used to evaluate the value of a student’s contributions and their interaction during wiki-based group work construction.

This study expands the typical method of counting student’s contribution in-group work assignment, by measuring quality of students’ contributions together with their interactions with group members.

We have proposed that the additional attributes can be used to obtain a better, more realistic assessment of the value of students’ contributions and their interactions.

The development of this framework seeks to benefit three main stakeholders: educators, students and software developers:

- For educators, an audited set of principles and guidelines will assist them in determining a student’s contribution and the value of their interaction during group work.
- For students, this study will provide guidance on how their group work will be assessed.
- For software developers, this study will provide insights into the features that could be included in the development of collaborative software used for assessment purposes.

Further work will involve trialling this assessment framework on a unit that uses wiki-based group work assignment. At this stage a postgraduate unit that focuses on digital marketing has been selected. Ethics has been sought to analyse the Wiki log files for students who completed this unit in 2012 and 2013. Analysis of the log files will be achieved by applying the first two dimensions of the framework, and will be reported in subsequent publications.

This will provide some insights into which attributes should remain or be removed from the framework.

6 Acknowledgement
The first author would like to thank the Directorate General of Higher Education of Indonesia for supporting his PhD study at Monash University Australia.

7 References


Matthews, D. & Schrum, L. (2003): High-speed Internet use and academic gratifications in the college


Trentin, G. (2009): Using a wiki to evaluate individual contribution to a collaborative learning project.
Solution Spaces

Nadia Kasto§, Jacqueline Whalley§, Anne Philpott§ and David Whalley†

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

†Educational Consultant
Auckland
New Zealand
geowah@yahoo.com

Abstract
This paper explores the idea of solution space in the context of novice programmers and code writing tasks. A definition for solution space is provided and an analysis of a series of code writing questions from a first year Java programming course’s practical programming tests is provided to measure the impact of solution space size on the difficulty of a code writing question. We found that as the solution space size increases so does the difficulty of the question and that despite relatively high solutions spaces we see a very limited set of these solutions as student responses. Finally we conclude with some conjectures about the possible causes for the trends that we have observed.

Keywords: novice programmers, code writing, assessment, task complexity.

1 Introduction

“Writing high quality readable text does not come easily to most young children. Many elementary teachers express frustration at the apparent poor written products emerging from their students.” (Beaglehole and Yates 2010). Similar themes have appeared in the computer-science education literature: students don’t know how to design programs, and they don’t know how to write programs. Soloway and Sophrer (1989) suggested that “students have difficulties in putting all the pieces together” and “many problems arise from structure composition problems”. Winslow (1996) supported this view stating that, “Study after study has shown that students have no trouble generating syntactically valid statements once they understand what is needed. The difficulty is knowing where and how to combine statements to generate the desired result”.

It is generally accepted by teachers that many students who are learning to write find the task easier if they are given a more open task. This premise is supported by Rogers’s learner-centered model of teaching (Rogers et al. 2013). When they are allowed to write about a topic of their own choice these students quickly decide what topic they would like to write about and how they will go about it. Some other students tend to flounder in such a large space and cannot decide how to get started. On the other hand if the students are directed to write on a specific topic set by the teacher, for example a grandparent's birthday, some find that the restricted scope makes the writing task easy for them while others have difficulty engaging in a task that provides them with such limited possibilities for writing. The reasons for finding a particular writing task difficult may include: a lack of personal experience- the students may never see their grandparents-, a lack of interest in the topic, a strong desire to write about a personally more motivating topic, a perceived absence of an audience for the finished product or a lack of the vocabulary needed to engage in the topic set. In effect, some find that a large solution space provides them with many opportunities and allows them to make choices that result in effective writing. Others find a large solution space daunting and have difficulty making productive choices. What effects do differences in the solution space of programming tasks have on the ability of novice programmers to successfully complete those tasks?

In programming there are many ways to tackle a fairly small problem, and different students can produce different solutions to the same problem. In a preliminary small scale study Carbone (2007) found that when students were given open programming tasks, tasks that had many possible ways to approach the problem and hence a large solution space, some students focused on a wrong aspect of the task or pursued a wrong approach as they lost track of the big picture. It seems reasonable to assume therefore that solution space has some influence on the difficulty of a novice programming task.

In a recent study that attempted to evaluate the difficulty of questions presented in final examinations the group of academics found it difficult to agree on the difficulty of questions (Simon et al. 2012). The degree of agreement between the academics in estimating difficulty was only 40% so the inter-rater reliability was poor. This finding indicates that it is difficult for educators to be objective in their estimations of difficulty of assessment items in computer programming. There is a tendency to both under and overestimate the difficulty of these tasks. Clearly there is a need for more objective measures of difficulty for novice computer programming tasks.

2 The solution space conjecture

Our conjecture is that the difficulty of code writing tasks, for novice programmers, is related to the size (and possibly other dimensions) of the solution space for a problem. We were also interested in whether or not the number of solutions provided by students, the students’
solution space, to a code writing problem is influenced in any way by the size of the problem’s solution space.

Luxtton-Reilly et al. (2013) investigated the variation in correct student solutions for problems. They defined three different types of variation: variation in structure, syntax (within a block) and presentation. For this research solution space is defined as the set of structurally or syntactically different solutions that provide a correct answer for a specific code writing problem. The addition of redundancies (for example, extra semicolons, empty if or else statements) have not been counted as additional solutions.

The notion that difficulty might be related to solution space size is perhaps not a surprising idea. Academics often consider the answers that we may get in response to a code writing assessment and write a rubric that will help accommodate those expected responses when marking the students’ answers. However to our knowledge the idea that solution space size may affect difficulty has not previously been tested.

3 The data set

The data for this work was gathered from a first semester Java programming course. The course was designed with the assumption that the students have no prior knowledge of computer programming. The course adopts a back to basics procedural approach (similar to that suggested by Reges (2006)) except that the learning is supported by an in-house micro-world called Robot World in the BlueJ IDE. Robot World was inspired by ‘Karel the Robot’ (Pattis 1981). For the majority of the course students do not write their own classes but instead learn to decompose their programs into methods. The advantages of using micro-worlds as a tool for teaching novice programmers are well documented. These advantages are that they:

- reduce the complexity of a language by providing a subset of a conventional language
- enable students to visualise the execution of the program, giving immediate feedback and assisting them in the debugging process (McIver and Conway, 1996)
- increase the focus on problem solving and algorithm design (Kölling, 1999),
- facilitate learning better than text-based (non-visual) systems (Dougherty, 2007).

It is for these reasons that the traditional back to basics approach was extended to include the micro world in a simplified learning IDE as the teaching environment for this course.

The eight code writing questions analysed in this study were selected from a series of summative practical programming tests held throughout the first semester of a first year Java programming course. Sixty student responses were analysed for each question. These students had given ethical consent for their data to be used and were representative of the entire cohort.

The questions analysed are provided in Appendix A. These questions were selected from a larger body of questions. These were questions which contained concepts that had been taught to the students but which were presented in a problem they had not seen before although they had seen examples that were variations (Thompson 2010) on the problem. An example scenario is provided below.

Question 5 asked the students to work out the length of two corridors and print out the length of the longer of the two corridors provided. The corridors could be of any length and may even be the same length. The students were provided with images of one possible starting scenario for the robot (Figure 1).

3.1 Determining the solution space

Solution space can be defined as the set of possible structural and syntactical permutations that provide a working solution without any discrimination of solutions due to the quality of the solution.

Two instructors developed a set of solutions to a set of first semester novice programming tasks. These sets of solutions formed the minimum solution spaces. It should be noted that each set is not necessarily the full set of all possible solutions as identifying the set of all possible solutions for a code writing task is an extremely complex problem and it becomes more problematic as the size and/or complexity of the code increases.

Even a relatively simple selection statement can generate several possible solutions. For this reason we define our solution space as at least a certain number of solutions; there may be other solutions which have not been identified.

The following discussion illuminates the way in which we have determined solution space size with an exemplar. Question 4 asks the students to write code that allows the...
robot to navigate through a spiral maze until they find a beeper at which point the robot should stop. Robots can only turn left. The students at this point have only learnt about while loops so the problem’s solution space only consists of solutions which contain a while loop. The solutions identified by the instructors which form the problem’s solution space are given in Table 2. This problem’s solution space is comprised of at least three candidate solutions and therefore has a size of at least three.

<table>
<thead>
<tr>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(isGroundClearAtRobot()) {</td>
</tr>
<tr>
<td>while(isSpaceInFrontOfRobotClear()) {</td>
</tr>
<tr>
<td>moveRobotForwards();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>turnRobotLeft();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>while (isSpaceInFrontOfRobotClear()) {</td>
</tr>
<tr>
<td>moveRobotForwards();</td>
</tr>
<tr>
<td>if (isRobotFacingWall()) {</td>
</tr>
<tr>
<td>turnRobotLeft();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>while (!isRobotFacingWall()) {</td>
</tr>
<tr>
<td>moveRobotForwards();</td>
</tr>
<tr>
<td>while(!isItemOnGroundAtRobot() &amp;&amp; (isRobotFacingWall())) {</td>
</tr>
<tr>
<td>turnRobotLeft();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Table 2: Solution space for Question 4

5 Results

Figure 2 shows that there is an obvious trend, for the questions we have examined, between solution space and question difficulty. The smaller the solution space the easier the students found the question.

![Figure 2: Solution Space Size (y axis left) and % correct answers (y axes right) by question](image)

6 Conclusions and future work

The Dreyfus Model of Skill Acquisition (Hunt 2008) suggests that novices copy solutions so if the teaching style provides patterns for solutions to a particular style of code writing problem then it is possible that the task maybe easier for the students regardless of the solution space size. Moreover, the students’ available solution space is likely to be influenced by factors such as the instructor’s teaching focus, previously seen code and the wording of the question itself.

For novice programmers the difficulty of a programming task tends to increase as the solution space increases. This relationship between difficulty and solution space could be used to estimate the difficulty of tasks set for students in computing labs or tests. A difficulty metric based on minimum solution space size should provide academics with a more consistent and reliable way of determining the probable difficulty of computing tasks designed for novice programmers. There is no doubt that a difficulty measure that is more accurate
than the 40% agreement about difficulty levels (Simon et al. 2012) achieved using the judgement of academics familiar with the teaching of novice programmers is desirable.

In natural language metrics the measures of difficulty have usually been grouped so that the results are presented in meaningful categories such as equivalent grade levels or difficulty levels. Computing tasks for novice programmers could also be grouped into categories of difficulty to provide a quick and easy estimation of the difficulty of a task. For example, for a first semester of programming a minimum solution space size of $1-4 = \text{easy}$, $5-7 = \text{medium}$ and $>7$ hard would probably be appropriate. This of course could be adjusted for subsequent courses and or standards for a course.

One of the limitations of this preliminary work is the need to increase the clarity and repeatability of the minimum solution space size calculation. It may be that a comparison of problem characteristics to typical solutions space sizes could shed some light on useful heuristics.

### References


## Appendix

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>For this question, the students are supplied with the method header. They are asked to complete the method body by writing a sequence of three statements to make the robot drop the beeper it is carrying, then move the robot forward one cell and turn the robot left once.</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>For this question, the students are supplied with the method header. They are asked to complete the method body so that the robot turns left then if there is no wall in the way moves forward one cell.</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>In this question, the students are provided with a robot in a cell that contains a number of beepers. The students are asked to write a method called <em>pickUpNBeepersCheckIfAll()</em> that takes an integer parameter, and makes the most recently created robot pick up that number of beepers from the beeper stack at its current location. You can assume that there are enough beepers in the stack for the robot to do this safely. The method should return true if the robot has picked up all the beepers at its current location, or false if there are still beepers on the ground.</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>Complete the method <em>navigateSpiral</em> that moves the robot through a spiral maze until it reaches a beeper. The spiral will always have 6 passages but they will be varying in length.</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>In this scenario there are interconnected two corridors, they are always connected at the same point (See Figure 1 for details). The length of each of the corridors changes randomly each time the robot world is created. A corridor number is specified by the row of the world that the corridor is in. The students are asked to: Write a program that measures the length of both corridors, and then displays the message <em>Corridor&lt;</em>&gt; is the longest. It is &lt;<em>&gt; long.</em> Where: &lt;<em>&gt; is the number of the longest corridor. &lt;</em>&gt; is the length of that corridor. If the corridors are the same length, the message should specify corridor 0.</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>This question asks the students to write a method called <em>walk()</em> that makes the robot walk through a door to reach a beeper. The door that it must walk through could be to the east or west or straight ahead […up…]. A door will always be present. The robot must only pass through the location in front of the door once.</td>
</tr>
</tbody>
</table>
In this scenario the robot starts off carrying 100 beepers, and there is also a pile of beepers at position (0, 0). The robot should pick up those beepers and count how many there are. Then the robot should draw a square using the beepers by dropping them. The length of the sides of the square in beepers should be the number of beepers picked up from position (0, 0). For example, if the robot picks up 5 beepers then it should make a 5 by 5 square.

This question asks the students to write a method called `advanceRobot()` that has two parameters a Robot and a distance to travel (the number of cells that the robot should advance). The robot should only be able to move if it is alive and if the distance to travel is positive if it is unable to move an appropriate exception should be thrown. If the robot encounters a wall before moving the full distance it should stop rather than crashing. The method should return true only if the robot moved the full distance.
Introductory Programming Courses in Australia and New Zealand in 2013 - trends and reasons

Raina Mason
Southern Cross Business School
Southern Cross University
Coffs Harbour NSW Australia
raina.mason@scu.edu.au

Graham Cooper
Southern Cross Business School
Southern Cross University
Coffs Harbour NSW Australia
graham.cooper@scu.edu.au

Abstract
This paper reports the results of a survey of 38 introductory programming courses in Australian and New Zealand universities, conducted in the first half of 2013. Results of this survey are compared with a survey conducted in 2010 on Australian universities and two other previous studies conducted in 2001 and 2003. Trends in student numbers, programming paradigm, programming languages and environment/tools used, as well as the reasons for choice of such are reported. Other aspects of first programming courses such as instructor experience, external delivery of courses and resources given to students are also examined.

The results indicate a trend towards the adoption of Python for Introductory Computer Programming courses and that this language is being used in a structured approach for programming. Introductory computer programming courses that focus upon an Object Orientated approach predominantly use Java.

Keywords: introductory programming, programming languages, programming environments, Australian university courses, New Zealand university courses, pedagogy, trends.

1 Introduction
Most Computer Science and Information Technology degree programs include at least one compulsory introductory programming course. Programming is generally perceived to be complex and difficult and these courses can suffer from high attrition rates and low levels of competency (McCracken et al. 2001). Debate continues on which languages, environments and paradigms should be used in a first programming course to maximise student success and motivation (Bloch 2000, Jenkins 2002, Pears et al. 2007, Dale 2005, 2006).

To establish the (then) current state-of-play in Australian and New Zealand universities, censuses were conducted in 2001 and 2003 (de Raadt et al. 2002, 2004) which reported on the languages, paradigms and environments/tools being used, the reasons for choice of language, student numbers (and the downwards trend) in each course, texts employed, instructor experience and the teaching of problem solving strategies.

In the latter months of 2010 a phone interview survey which repeated the previous two surveys with minor changes was performed with a large sample of 44 programming courses, across 28 Australian universities (Mason et al. 2012). Longitudinal trends in languages, tools and paradigms were identified, as well as reported reasons for such changes over the 10 year period since the survey was initially conducted. The 2010 survey showed Java as the most popular language, followed by Python and then C. “Pedagogical benefits” was the most common reason for the choice of language, followed by “Relevance to industry/marketability to students”. The procedural paradigm was most often used for teaching, and fewer participants (20% compared to 43% in 2003) were choosing to use only text editors and command-line compilers rather than IDEs or other tools.

In early 2013 the survey was repeated in an online survey format, with Australian and New Zealand universities invited to participate. Details about the interview questions and the methodology of the study are described in the next section, followed by results and discussion of the implications for teaching introductory programming.

2 Methodology

2.1 Recruitment of participants
The list of participants from the 2010 study was used as a starting point for contacting potential participants. An email was sent to each previous participant inviting participation in the 2013 study. As the survey was to be conducted online, rather than by telephone interview which imposed cost and time-zone difference issues, New Zealand universities were included in this study.

University websites were used to identify potential participants from New Zealand and invitations were sent either directly to potential participants, or to administrative staff responsible for those programs. A general invitation to participate was sent to the SIGCSE-Australasian mailing list and the SIGCSE list for the attention of the Australian and New Zealand members.

The online survey was open from mid-April to mid-July 2013, when it was closed and the results were downloaded and analysed.

2.2 Questions
For all questions, the terminology “course” was used for the basic unit of study that is completed by students towards a degree, usually studied over the period of a semester or session in conjunction with other units of...
study ("courses"). This terminology was used to maintain consistency with the previous studies.

Large portions of the 2013 survey questionnaire were drawn from the previous studies including questions about language and paradigm choice, programming environment/development tools, instructor experience, reasons for choice of language, and perceived difficulty of the language and environment (if one is used).

Additional questions were added to ascertain the relative importance of each reason given for language and environment choice. It was anticipated that there may be a relationship between the choice of language and environments and the reasons for these choices. Instructors were also asked how useful the language was for teaching the fundamental concepts of programming.

A final section asked instructors to identify what he or she considered to be the 3 most important aims of the introductory programming course. Other general interest questions were asked regarding whether the course was offered in external mode, and what resources were provided to students.

3 Results and Discussion
The results of this study are reported below, with comparison to the previous three studies where applicable.

3.1 Universities and Courses
The number of courses covered in the 2013 study was fewer than each of the other three studies. Forty-eight courses from twenty-nine Australian and New Zealand universities participated, however eight participants failed to progress through the study and these surveys were not analysed. A further two participants gave details of the course and student numbers but did not answer questions on programming languages or environments. Some participants answered most but not all questions. This has been indicated in the results and discussion where necessary. Participants were asked for their course codes and universities, so matching could be performed with previous surveys, where necessary. This also eliminated possible duplication.

3.2 Student Numbers
Comparison of the 2001, 2003, 2010 and 2013 course participation and reported numbers of students are given in Table 1.

From 2010 to 2013 there appeared to have been a 50% increase in the average number of students per course, bouncing back to pre-2003 levels. In case this was an institution effect (i.e. larger institutions participating in this survey than in 2010), where possible, courses that participated in the 2010 study were directly compared with the same courses in the 2013 study. Comparing courses that participated in both studies gave an increase from a 2010 mean of 198 students per course to a 2013 mean of 253 students per course - a 27.8% increase in students over 4 years. The apparent increase in student numbers is consistent with the trends in ACS data to 2010 (latest figures) which show a 4.5% increase in enrolments across the sector from 2009 to 2010 (ACS, 2012).

This is good news for the ICT industry which is predicting a significant shortfall of suitably educated and skilled ICT professionals in the near future (DEEWR 2011).

3.3 Languages
3.3.1 Choice of Language(s)
One of the main areas of interest to this study was the language(s) being used in these introductory programming courses. Instructors were presented with a choice of languages used in the previous three studies and asked to indicate which they used in their courses, as well as offered a space to indicate other languages.

In the 2013 study, a total of 12 languages were used in first programming courses. The majority (33) of courses used one language throughout the first programming course (Table 2). When more than one language was used, the generic approach adopted was for one language to be used initially and then another language added (while keeping the first). In only one case the course was segmented into learning different languages consecutively.

<table>
<thead>
<tr>
<th># of languages</th>
<th>2010 courses</th>
<th>2013 courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3-6</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Comparison of number of languages/course

<table>
<thead>
<tr>
<th>Language</th>
<th>Courses</th>
<th>%age</th>
<th>Weighted by students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>12</td>
<td>27.3%</td>
<td>26.9%</td>
</tr>
<tr>
<td>Python</td>
<td>12</td>
<td>27.3%</td>
<td>33.7%</td>
</tr>
<tr>
<td>C#</td>
<td>4</td>
<td>9.1%</td>
<td>4.8%</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>6.8%</td>
<td>8.6%</td>
</tr>
<tr>
<td>Javascript</td>
<td>3</td>
<td>6.8%</td>
<td>10.3%</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>3</td>
<td>6.8%</td>
<td>1.4%</td>
</tr>
<tr>
<td>C++</td>
<td>2</td>
<td>4.5%</td>
<td>3.0%</td>
</tr>
<tr>
<td>Ada</td>
<td>1</td>
<td>2.3%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Haskell</td>
<td>1</td>
<td>2.3%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Matlab</td>
<td>1</td>
<td>2.3%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Scribble</td>
<td>1</td>
<td>2.3%</td>
<td>5.6%</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
<td>2.3%</td>
<td>0.5%</td>
</tr>
</tbody>
</table>

Table 3: 2013 Languages

The programming languages used by the participant courses are shown in Table 3. Languages are presented by...
number of courses, percentage of courses, and weighted by student numbers. Note that the “courses” column will add to more than 38 courses, as some courses used more than one language.

The top three languages in the first half of 2013 (in order) were Java, Python and C# (by courses) and Python, Java and Javascript weighted by students (Figure 1).

![Figure 1: Programming Languages in 2013 by courses and students.](image)

<table>
<thead>
<tr>
<th>Language</th>
<th>2001</th>
<th>2003</th>
<th>2010</th>
<th>2013</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>40.4%</td>
<td>40.8%</td>
<td>36.4%</td>
<td>27.3%</td>
<td>-9.1%</td>
</tr>
<tr>
<td>Python</td>
<td>0%</td>
<td>0%</td>
<td>13.6%</td>
<td>27.3%</td>
<td>13.7%</td>
</tr>
<tr>
<td>C#</td>
<td>0%</td>
<td>0%</td>
<td>9.1%</td>
<td>9.1%</td>
<td>0%</td>
</tr>
<tr>
<td>C</td>
<td>7%</td>
<td>12.7%</td>
<td>11.4%</td>
<td>6.8%</td>
<td>-4.6%</td>
</tr>
<tr>
<td>C++</td>
<td>0%</td>
<td>0%</td>
<td>2.3%</td>
<td>6.8%</td>
<td>4.5%</td>
</tr>
<tr>
<td>Javascript</td>
<td>0%</td>
<td>0%</td>
<td>9.1%</td>
<td>6.8%</td>
<td>-2.3%</td>
</tr>
<tr>
<td>Matlab</td>
<td>14%</td>
<td>11.3%</td>
<td>7%</td>
<td>4.5%</td>
<td>-2.5%</td>
</tr>
<tr>
<td>Alice</td>
<td>0%</td>
<td>1.4%</td>
<td>2.3%</td>
<td>2.3%</td>
<td>0%</td>
</tr>
<tr>
<td>Haskell</td>
<td>0%</td>
<td>4.2%</td>
<td>0%</td>
<td>2.3%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Ada</td>
<td>1.8%</td>
<td>0%</td>
<td>2.3%</td>
<td>2.3%</td>
<td>0%</td>
</tr>
<tr>
<td>Processing</td>
<td>0%</td>
<td>0%</td>
<td>4.5%</td>
<td>0%</td>
<td>-4.5%</td>
</tr>
<tr>
<td>Fortran</td>
<td>0%</td>
<td>1.4%</td>
<td>2.3%</td>
<td>0%</td>
<td>-2.3%</td>
</tr>
</tbody>
</table>

Table 4: Longitudinal language comparison – courses

Table 5: Longitudinal language comparison – students

<table>
<thead>
<tr>
<th>Language</th>
<th>2001</th>
<th>2003</th>
<th>2010</th>
<th>2013</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>0%</td>
<td>0%</td>
<td>19.5%</td>
<td>33.7%</td>
<td>14.2%</td>
</tr>
<tr>
<td>Java</td>
<td>43.9%</td>
<td>44.4%</td>
<td>39%</td>
<td>26.9%</td>
<td>-12.1%</td>
</tr>
<tr>
<td>Javascript</td>
<td>0%</td>
<td>0%</td>
<td>1.5%</td>
<td>10.3%</td>
<td>8.8%</td>
</tr>
<tr>
<td>C</td>
<td>5.5%</td>
<td>10.6%</td>
<td>11.9%</td>
<td>8.6%</td>
<td>-3.3%</td>
</tr>
<tr>
<td>C#</td>
<td>0%</td>
<td>0%</td>
<td>8.2%</td>
<td>4.8%</td>
<td>-3.4%</td>
</tr>
<tr>
<td>C++</td>
<td>15.2%</td>
<td>18.7%</td>
<td>4.9%</td>
<td>3%</td>
<td>-1.9%</td>
</tr>
<tr>
<td>Matlab</td>
<td>0%</td>
<td>1%</td>
<td>1.3%</td>
<td>1.7%</td>
<td>0.4%</td>
</tr>
<tr>
<td>Haskell</td>
<td>8.8%</td>
<td>6%</td>
<td>0%</td>
<td>1.7%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Ada</td>
<td>1.7%</td>
<td>0%</td>
<td>0%</td>
<td>1.7%</td>
<td>1.7%</td>
</tr>
<tr>
<td>VB/VB.NET</td>
<td>18.9%</td>
<td>16.4%</td>
<td>5.2%</td>
<td>1.4%</td>
<td>-3.8%</td>
</tr>
<tr>
<td>Alice</td>
<td>0%</td>
<td>0%</td>
<td>0.9%</td>
<td>0.5%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>Processing</td>
<td>0%</td>
<td>0%</td>
<td>5.3%</td>
<td>0%</td>
<td>-5.3%</td>
</tr>
<tr>
<td>Fortran</td>
<td>0%</td>
<td>0.7%</td>
<td>3.9%</td>
<td>0%</td>
<td>-3.9%</td>
</tr>
</tbody>
</table>

Figure 3: Longitudinal trends – top 3 languages of each year by students.

3.3.2 Reasons for choice of language

In the previous studies instructors were asked about the reasons for their choice of language. The two most common reasons given in both the 2001 and 2010 studies were “industry relevance/marketability to students” and “pedagogical benefits”. The 2010 study saw shifts in the frequency of some of the reasons given, with, for
example, industry relevance/marketability declined from 56.1% to 48.8% and pedagogical benefits increased from 33.3% to 53.5%.

The 2001 and 2010 surveys identified the reasons for the choice of language, but did not distinguish between the importance of these reasons. For example, an instructor may have indicated that “structure of degree/department politics” and “platform independence” were two reasons for their choice of language. One of these reasons may have been very important in their choice, and the other only slightly important. The 2001 and 2010 studies did not distinguish between the importance of these reasons and only counted frequencies of given reasons.

The 2001 and 2010 surveys identified the reasons for the choice of language, but did not distinguish between the importance of these reasons. For example, an instructor may have indicated that “structure of degree/department politics” and “platform independence” were two reasons for their choice of language. One of these reasons may have been very important in their choice, and the other only slightly important. The 2001 and 2010 studies did not distinguish between the importance of these reasons and only counted frequencies of given reasons.

The reasons offered for choices were those offered by the participants in the 2010 survey as well as a space for ‘Other’. Figure 4 shows the frequency for reasons given for choice of programming language - not weighted by importance of the reason.

Figure 5 presents the frequencies for identifying a reason for choice of language as “very important”.

The first three ranks for “very important” reasons are: 46% for “Pedagogical benefits of the language”, 44% for “Platform independence”, and 36% for “Relevant to industry”. A second analysis was conducted whereby the frequencies for identifying a reason as either “important or very important” was considered. The rank order of reasons between these two methods of analysing importance are not the same. In this case the first three ranks (noting that there was a tied first rank and tied third rank) for “important or very important” reasons are: 79% for “Pedagogical benefits of the language”, 79% for “Relevant to industry”, 67% for “Platform independence and 67% for “Availability/ cost to students”.

It should be noted that both methods of analysis return “Pedagogical benefits of the language” as a first rank.

Comparison of Python and Java

Given that Python has had a large increase in popularity, with a corresponding drop in popularity for Java, and given that these two languages represent over 60% of students in the survey, it was decided to make direct comparisons between the reasons for choice of Python and Java. Note that not all participants who use Python and Java have given reasons for their choice.

The first method of analysis for this purpose was to identify the reasons which all participants identified as a reason for the choice of language (varying importance being either slightly important, important or very important).

Python: All of the Python-using participants gave the following reasons for their choice (varying importance):

- Availability/Cost to students
- Easy to find texts
- Extensions/Libraries available
- Platform independence

Java: In contrast, all of the Java-using participants gave the following reasons for their choice (varying importance):

- Object-Oriented Language
- Online community/Help available
- Relevant to industry

It is interesting to note that there is an absence of overlap between these two sets of reasons. That is, the set of reasons which all instructors using Python offered for their choosing of Python is mutually exclusive to the set of reasons which all instructors using Java offered for their choosing of Java. Although Java is free for students and platform independent, these reasons appear to be more important to those choosing Python. Although Python is an object-oriented language, those looking for an object-oriented language are tending to choose Java. See Section 3.4 for more information about paradigm choices and the relation to language choice.
Note, however, that this analysis includes identification of reasons that are ‘slightly important’. Excluding the ‘slightly important’ reasons to focus upon the combined set of important / very important reasons yields the data presented in Figure 6 showing the set of important/very important reasons given for choice of either Java or Python.

From Figure 6, the important/very important reasons that return at least an 80% selection rate for choice of Python are:
- 91% Availability / Cost to students [Java 50%]
- 91% Pedagogical benefits [Java 75%]
- 82% Platform Independence [Java 75%]
- 82% Easy to find texts [Java 58%]

The important/very important reasons that return at least an 80% selection rate for choice of Java are:
- 92% Object-oriented language [Python 18%]
- 92% Relevant to industry [Python 73%]

It should be noted that Python is an object oriented language but can be used in a structured way with no necessity to discuss objects (at an introductory level). In comparison, Java is difficult to teach without providing some class structure, either by using an environment such as BlueJ or Greenfoot or by providing students with skeleton code and getting them to fill in the blanks.

The choice of language appears to have not been done at a mere surface level, but rather, with deep consideration as to how the language is to be used strategically with respect to presenting programming activities to students.

3.3.3 Perceived difficulty and usefulness to teach fundamental concepts

Participants were asked to indicate how difficult they believed their chosen language was for novice students, on a Likert scale of 1 - 7 where 1 was ‘very easy’ and 7 was ‘very difficult’. The medians of the results are given below in Figure 7. Note only languages where answers have been given by at least 2 participants have been included.

From these results, Java is perceived as more difficult for novices than Python. C is considered the most difficult for novices.

Regardless of whether or not the various languages really do exhibit these relative levels of difficulty, instructors are indicating that they perceive these relative levels of difficulty to exist, and this may be a factor of consideration in their choice of language.

Participants were also asked about the perceived usefulness of their language for teaching the fundamental concepts of programming, on a 7-point Likert scale where 1 was ‘very useless’ and 7 was ‘very useful’. The medians of their answers are given below in Figure 8. All languages, other than Javascript, are reported at about ‘6’ on the 7 point Likert scale. Javascript is reported at ‘4’.

3.3.4 Reasons for changing language

Respondents were also asked to rank reasons for which they might consider changing language in their course. Figure 9 presents the frequencies for identifying the first rank reason for which participants might consider changing language. ‘Pedagogical benefits’ accounts for close to half of all first rank preferences (47%) and attracts about 3 times as many nominations as the next most common factor, which is ‘Relevant to industry’ (15%).

A second analysis was conducted whereby the frequencies for identifying a reason in any of the top three
reasons for considering a change of language was considered.

While there were some slight variations to the rank order listings of some of the less common reasons, the first rank remained as ‘Pedagogical benefits’ (68%) and the next most common factor was again ‘Relevant to Industry’ (44%).

Figure 9: Reasons ranked in top 3 for considering change of language in their course.

3.4 Paradigm taught

Figure 10 presents trends for use of each paradigm over the set of four studies from 2001 to the current. Three aspects are apparent. The continuing dominance, and increasing rise of a procedural approach, the moderately low use of an object orientated approach, and the very low frequency use of a functional approach.

Although not reflected in Figure 6, in 2013, as in 2010, some instructors commented that they had chosen ‘procedural’ but introduced some aspects of object-oriented programming at the end of the course.

Figure 10: Trends in paradigms taught (4 studies)

Java vs Python: what paradigm is being used? Table 6 shows language (Java or Python) vs. paradigm chosen.

<table>
<thead>
<tr>
<th>Language</th>
<th>Procedural</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Python</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6: Paradigm by Language – Java and Python

An analysis was conducted comparing the language chosen (Python versus Java) by the preferred paradigm used for teaching (Procedural versus Object Oriented). This returned a statistically significant difference (Fisher exact test: p < 0.001). Instructors who reported object oriented approaches to their introductory programming courses were predominantly using Java. Conversely, instructors who reported procedural approaches to their introductory programming courses were predominantly using Python.

Despite Python being an object orientated language, instructors are choosing it and then using it in a procedural/structured way. Java can also be used in this way but it is more difficult unless some additional strategies are included (such as using the BlueJ environment). It appears that the objects-first instructors, also influenced by industry-relevance, are drawn towards Java and the procedural-first instructors (who may also wish to introduce objects later in the course, in the same programming language) are selecting Python.

3.5 Instructor Experience

Participants were asked to indicate their level of experience in teaching introductory programming, and as with the other studies in 2003 and 2010, there was a large range of experience. Four participants had less than 2 years, while two others reported over 30 years of experience. The majority had between 10 and 20 years of experience. This is consistent with the 2010 survey, where participants had a mean of 12.3 years of experience with standard deviation of 7.3 years.

3.6 IDEs and Tools

3.6.1 Choice of IDE/tools

An environment is used in most courses (77.8%) and by most students (69.7%). A significant proportion (22.2%) of the courses surveyed did not use any environment apart from text editors and command-line compilers. This is a similar figure to the 2010 results, and much fewer courses with no environment than in 2001 and 2003.

Of the courses that did use environments, Visual Studio was the most popular IDE at 15.6% of courses. Eclipse was used with 11.1% of courses. Idle and BlueJ followed with 8.9% each, and Netbeans at 6.7%. The remainder of the courses used Alice, Greenfoot and Quincy (all at 2.2% of courses), one ‘in house web-based environment”, and various other tools. Figure 11 indicates the percentage of students exposed to each of the major environments.

Figure 11: Environments by percentage of students
There are specific relationships between languages and environments, so comparison between different environments is awkward. For example, Idle is an IDE with Python which comes bundled with the language, while Netbeans and BlueJ are used with Java. Several environments, such as Visual Studio, can be used with multiple languages. Nevertheless, an approach that focuses upon why any specific environment was selected may provide insight into the dynamics and attributes of an environment that motivate their selection and use.

3.6.2 Reasons for choice of environment
The details of which IDE has been used in which language and for which reason is omitted due to space restrictions, but the primary reasons (and motivations) for selection and use of an IDE are presented.

The five most frequent reasons provided for selecting an IDE (not weighted by importance) which each scored at least 80% were: 88% pedagogical reasons, 88% visual cues/debugger, 85% uncomplicated/ease of use, 82% availability/cost to students, 82% student motivation (see Figure 12).

Analysing on the basis of reasons that have been identified as ‘very important’ yields the four most frequent responses where each scored at least 30%: 33% graphical user interface, 30% visual cues/debugger, 30% supports OO paradigm, and 30% pedagogical benefits.

3.7 Other Aspects of the course
3.7.1 External delivery
Of the 34 courses that answered this part of the survey, the majority (65%) indicated that they do not offer their course via distance or external mode, i.e. a mode where students are not required to attend regular lectures, workshops, labs or tutorials.

3.7.2 Resources given to students
Courses, whether offered externally or not, have various resources provided to students. Below in Figure 14 are the frequencies of resources reported by participants:

Figure 12: Reasons for choosing environments

3.6.3 Difficulty of environment
Instructors were asked to indicate how difficult they believed the environment was to use for themselves, and for novice students. The results indicate explicitly that instructors perceive students to have more difficulty with an environment than the instructors. This is consistent with the 2010 study indicating the same effect for language. Comparative difficulty is shown below in Figure 13, where 1 is “very easy” and 7 is “very difficult”.

Note that if a student is finding the use of an environment “somewhat difficult” and the language “somewhat difficult”, they may not have the cognitive resources available to problem solve, or develop algorithmic thinking (see Section 3.8).
3.8 Aims of an introductory programming course

3.8.1 Aim of the course – all languages
Participants were asked what they considered were the three most important aims of an introductory programming course. Answers varied but some themes became apparent. Around half of the participants indicated that nurturing algorithmic thinking was one of the main aims of the introductory course, closely followed by giving student an introductory experience of what it was like to program, and ‘learning fundamental concepts’. Interestingly, problem-solving and learning syntax did not appear in the top 3 aims. The themes for which at least two instructors agree are given below in Figure 15.

There were numerous other themes with only one instructor identifying each: basic writing skills, programming proficiency, teaching students to program, basic tools of programming, breadth of paradigms, computing literacy, conceptual models, differentiation of students, real industry-type experience, planning skills, see results, attention to detail, clarity of expression, modification of code, and programming achievement.

3.8.2 Aims of the course – Java vs Python
The reasons given for the choice of environment were compared for courses using Java and courses using Python and the results are presented in Figure 16. Visual inspection indicates that ‘algorithmic thinking’ was the most important reason for selection of an environment for use with Python, but this had relatively little influence in the selection of an environment for use with Java.

There were several factors that contributed relatively higher for the selection of an environment for use with Java compared to Python, including:
- fundamental OO concepts,
- fundamental concepts,
- fundamental constructs and
- confidence building.

Figure 15: Aims of the introductory course

Figure 16: Aims – Java vs Python

4 General Discussion
Two languages currently dominate use in introductory programming courses in Australia and New Zealand. Java has been the most popular language for this purpose since at least 2001 until the present, where it has now fallen to second most frequently used language (as measured by number of students receiving the language). The majority of instructors who use Java have indicated that the primary reasons for their choice of Java have been for its industry relevance and object oriented paradigm.

The language that is now presented to the highest number of students in Australasia (based upon this study) is Python. Python is a relatively new language, and did not even appear in the 2001 and 2003 censuses of introductory programming courses in Australasia, which the current study seeks to broadly repeat. Python has delivered a substantial and sustained impact upon university delivered courses in introductory programming, with Python rising from nothing to top rank in ten years.

The majority of instructors who use Python have indicated that the primary reasons for their choice have been student focussed. This includes pedagogical benefit to facilitate student learning but also other aspects to make life easy for students, through minimising cost and maximising platform independence and access to learning support in the form of textbooks.

The two factors that have been of primary importance for language selection since the 2001 study (de Raadt et al. 2002) have been industry relevance and pedagogical benefits. This is still the case, but whereas in 2001 the reason ‘pedagogical benefits’ was second to industry relevance, it has now risen in relative importance to be the most common reason for language selection. Instructors, when queried about what would motivate them in the future to change language, have indicated a weighting towards pedagogical benefits 3 times more commonly than the second most important factor...industry relevance.

The two factors of pedagogical benefit and industry relevance do not necessarily work together in harmony. A language that is “ideal to industry” will not necessarily be a language that also offers “pedagogical benefits”. The vice versa is also true; a language that offers high
“pedagogical benefits” will not necessarily be highly relevant to industry. While it may be logically possible for a language to score highly on both of these attributes, it does not appear to be reflected in the reasons currently offered for selection of a language for introductory programming. There is an apparent tension in a dichotomy of Java being selected for OOP and industry relevance, while Python is being selected for ease of student learning and overall uncomplicated experience.

The heightened emphasis given to pedagogical benefits is also demonstrated in instructors’ responses regarding selection and use of environments. Although there is a relatively wide range of environments, with sometimes complex relations to a range of languages, the motivations and reasons for selection align broadly to those identified for language selection.

Some of the primary reasons for adopting an IDE are again associated with pedagogical benefits. Indeed, several of the reasons that were highly rated, such as ‘GUI’ and ‘uncomplicated/ease of use’ have, for theoretical reasons, been identified through Cognitive Load Theory (Sweller, 1999) to be likely mechanisms to reduce a student’s cognitive load, and thus facilitate learning.

There is a clear and continuing trend for instructors of introductory programming courses to be mindful of aspects of their student’s experiences in the context of learning computer programming. This always involves aspects of sitting at a computer, using an interface to navigate and operate upon elements of code, syntax and structure.

As a more complete understand of the dynamics of student learning, thinking and program construction is obtained, and as these feed into future computer program interfaces and architectures, it is anticipated that instructors may continue to enhance their focus upon consideration of student (learner) focussed aspects of introductory programming. These may continue to play an important role in the selection of languages and environments for introductory programming and represent areas for further research.

5 Acknowledgements

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

6 References


Unblocking the pipeline by providing a compelling computing experience in secondary schools: are the teachers ready?

Catherine Lang, Annemieke Craig and Gail Casey
La Trobe University, Melbourne; Deakin University, Geelong
Victoria, Australia.

c.lang@latrobe.edu.au; acraig@deakin.edu.au; g.casey@deakin.edu.au

Abstract
The decline of student interest and participation in computing degrees at university is affecting the stability of computing as a stand-alone discipline in universities. Research indicates that the decline begins in secondary schools. This paper describes an outreach program that was funded by an Australian Council of Deans of ICT grant. The researchers, acknowledging the time-poor nature of teachers’ work and that some of them are not trained in the computing discipline, developed curricula and provided resources and student helpers to enable secondary school teachers to deliver a student-centred unit of work. This unit focused on a four week program with students developing applications for android phones. The program was delivered in three schools by four teachers and produced mixed evaluation results. In one school the number of students taking ICT the following year increased significantly, this was not reported in the other two schools. Our findings show that even when teachers are provided with resources and artefacts, not all are prepared to deliver a fully student-led classroom experience. We ask “are the teachers ready?” to embrace transformational pedagogies using ICT in the classroom. In this case study we can say some are, but some are not. We also note that the technical issues within school networks hamper the ability of teachers to provide compelling computing experiences to students. Our recommendation for future implementations of the program is to provide teachers with more background on the benefits of a student-centred classroom approach before beginning this four-week unit of work.

Keywords: Broadening participation, Diversity, Computing Education, Outreach, transformational pedagogies.

Introduction
Most computing educators are aware of the downturn of student enrolment numbers in both higher education computing courses and senior secondary school courses in Australia. For example, the Australian Computer Society report that less than 3% of all university students are studying undergraduate computing degrees (ACS 2012). This is not an issue limited to Australia but reported in other westernised nations such as the USA and UK (Durando, Wastiau and Joyce 2009, Mentornet 2012). In the state where the program described in this paper was carried out the decline in the number of students studying the computing discipline in senior secondary school is alarming. The total number of students studying final year units has decreased by 75% since 2001, and the lack of gender diversity within this cohort is equally dramatic, for example in 2001 there were 5879 female students enrolled in the final year computing exams, and in 2011 only 643 (VCAA 2011). The trend of declining enrolments is similar in student selection of university computing courses, a decline of 67% in the same time period (VTAC 2013). These statistics are indicative of a decline in student perception of ICT courses being a valid and relevant component of a future career (Lang 2012).

Research has shown that a positive and engaging experience with ICT in the classroom can spark student interest and desire to pursue this discipline (Lang 2010; Fisher, Lang, Craig and Forgasz 2012) and that engaged and enthused teachers are critical to the running of a successful program (Guzdial and Ericson 2012; Ericson Fisher, Lang and Forgasz 2013). This literature informed the design and delivery of the outreach program that is the focus of this paper. In particular our program was designed to relieve the pressure of time-poor teachers in that the curricula was written to encourage a student-led peer support classroom model of delivery. This allowed teachers to be relieved of the pressure to be the computing expert. Secondly we provided the physical equipment needed, a class set of android phones and a curricula that allowed students to create of a meaningful product – a series of mobile phone games adapted from a popular textbook (Wolber, Abelson, Spertus and Looney 2011). The third unique aspect that contributes to building strong pathways between students in secondary schools and those in universities was to provide each class with a

Copyright (c) 2014, Australian Computer Society, Inc. This paper appeared at the 16th Australasian Computer Education Conference (ACE 2014), Auckland, New Zealand, January 2014. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 148. J. Whalley and D. D’Souza, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.
student facilitator, a current undergraduate student, who could assist the teacher in delivery of the program and also act as an informal role model to the school students. Furthermore, we purposely recruited female students to be our facilitators to debunk the myth that computing is a boys only subject area.

In the following sections we provide further information on the background to the study, a description of and justification for our method of delivery and curriculum development. This is followed by the findings from four instantiations of the outreach activity. The paper concludes with a discussion about the impact and future direction of future outreach activities, as well as recommended modifications to our model to ensure that future implementations avoid the same pitfalls.

1 Background

Technology has become more pervasive in every part of life in the 21st century and computing qualifications offer students dynamic career opportunities to work in any sector. Ironically as stated in the introduction student enrolments in these courses are in decline. Prior research into factors that influence student course choices (Lang 2012) and experience with outreach programs to address gender diversity (Lang, Craig, Fisher and Forgasz 2010) emphasised the importance of enthusiastic teachers to student course choices. Australia’s secondary education curriculum is currently undergoing a review and it is suggested that this will equip students with relevant computing skills and knowledge, offer a compelling learning experience that will inspire them to pursue ICT computing skills and knowledge, offer a compelling curriculum development. This is followed by the findings on the background to the study, a description of and justification for our method of delivery and curriculum development. The paper concludes with a discussion about the impact and future direction of future outreach activities, as well as recommended modifications to our model to ensure that future implementations avoid the same pitfalls.

1. Deliver a compelling computing experience to secondary school students via the creation of a meaningful product and seeing it through to completion (e.g. a game for android phones).

2. Provide time-poor teachers with support materials and necessary artefacts to deliver the program (student-led curriculum, mobile phones, and support in setting up the program).
3. Provide intentional role modelling by placing undergraduate students in the classroom to work with the teacher to deliver the program. We deliberately recruited female students to promote diversity.

In doing so the program addressed the issues of general non-participation of students in computing and lack of diversity of the cohort. The second issue addressed was the school teacher’s lack of time to prepare new materials and lack of access to up to date curricula and resources.

Funded by an Australian Council of Deans of ICT Engagement Grant (2012) we were able to finance our research and purchase the necessary class sets of mobile phones. After gaining ethics approval from our institution we invited school teachers to a consultation workshop to determine their acceptance of what we believed was a meaningful activity that would align with the current school curricula. The outcome of this session was the brief to develop new course materials that allowed for the student-led classroom environment, and that was extensible to allow it to fit in as one module of four weeks duration, with an average of 3 classes each week that was suitable for delivery to years 9, 10 or 11 students.

This resulted in a module of work that could be delivered over 4 weeks depending on the timetabled classes (provided in Appendix 1). We knew that the schools had an average of 3 lessons a week for computing electives, with each lesson being typically 40 to 50 minutes duration. The module of work was focused on fostering creativity using the web based application hosted by MIT. It provided teachers with preparation instructions such as to install and pre-test the software as recommended on the MIT website, create an online sharing space for the project or use emails and set up the necessary Google mail accounts. The module also suggested assessment areas linked to the required learning areas of Visualising Thinking, Creating for Communicating and Design, Creativity and Technology. Lessons were structured around discussion, student-centred sharing for tips and hints, written tasks for peer and self-assessment, and reflection. Worksheets were provided, a list of online resources, helpful websites and a reminded that the activities were written to be student centred and that the teacher was not expected to have a complete understanding of all the tutorials and skills, acknowledging their time-poor situation. Week 1 activities were built around the theme of “Exploring”, Week 2 was “Learning by sharing”, Week 3 was “Creating” and the final week focused on showcasing student work. Assessment tasks were constructed to reflect the student-centred approach and the scaffolded learning. Tasks 1 and 2 were focused on the students’ ability to help others, Task 3 on Knowledge building and Task 4 was collaborated group work and sharing with a lower grade level (we suggested Year 7 students).

The program was delivered in the second half of 2012 in three different schools, one of which repeated the program in semester 1 2013, so the evaluation is from four instantiations. There were also four teachers involved with the delivery because the school that delivered it twice had a different teacher for the class in 2013.

The timeline of the project was from June 2012 through to May 2013. Tools were also developed for collection of data and evaluation. Student helpers were employed and instructed to write a weekly blog at the end of each lesson they attended. We developed a teacher pre-survey and conducted post program interviews with the teachers. We were particularly interested in determining how they implemented the program and what issues they encountered. Given the short time-frame of the research grant it was not feasible to get parental permission to conduct student surveys. Teachers reflections and their reporting on student acceptance of the program was deemed to be sufficient in this instance, coupled with the observations of the student helpers, we believed we could obtain a satisfactory assessment of the effectiveness of the outreach activity.

4 Findings

The teacher interviews and student helper blogs provide a good insight into the effectiveness of this program. We provide a summary of these in the following sections.

4.1 Teacher Feedback

<table>
<thead>
<tr>
<th>School</th>
<th>Characterisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Male teacher. Independent School. High level Year 10 Multimedia Elective (teacher reported). Six apps used. 18 students (6 females)</td>
</tr>
<tr>
<td>B</td>
<td>Male teacher. Government School. Low level Year 11 class (teacher reported behavioural and struggling ESL students). Three apps used. 20 students (4 females)</td>
</tr>
<tr>
<td>B2</td>
<td>Male2 teacher. Teacher was a replacement for female teacher who started the program and had delivered 2 apps. Year 10 elective. 18 students (3 females)</td>
</tr>
<tr>
<td>C</td>
<td>Male teacher. Government School. Year 10 Game Design elective. 25 students (3 females)</td>
</tr>
</tbody>
</table>

Table 1: Summary of each class

In each of the follow up interviews it was obvious that the success of the program was tied very closely to whether the teacher followed the modules as they were designed. However, in each case the teacher said that they would use the unit in the following year. The two teachers who used the program as designed (B and C) were the most positive in their feedback. Teacher A did not use the mobile phones provided, admittedly there was a glitch in delivery and they did not arrive until mid-way through week 1 of the course. He also did not feel empowered to direct the university helpers to conduct any specific help or activity, despite having met them before the start of the unit.

*Uni students walked around the class once or twice per week. They were mainly viewers. Did help some girls but were not used to being in the classroom. Probably needed more structure as to who and how to target. (Teacher A)*

Teacher B2 also did not use the phones initially because he was afraid that students would break them. Teachers
A, B and B2 reported technical difficulties, and slow internet connections:

School system was a bit slow on the Internet - google problem. This was frustrating. Needs to be on the school servers definitely. (Teacher A)

One obstacle - internet in the room failing frequently. Internet based software was an issue. For bits of periods and sometimes it was slow. (Teacher B2)

Teacher B saw it as an advantage that students had to work through problem solving own laptops:

The students can actually produce an App that they can put on their phone. A real link between what you are doing and what happens in the real world. (Teacher B)

This teacher was the most enthusiastic about the program and reported that he delivered a professional development overview to his colleagues in an after school meeting.

Fantastic course outline. I could pick it up with four weeks planned... All I had to worry about was getting familiar with the programming.

Teacher B commented that he did not usually do group work and found this a useful challenge. He structured it by allocating two minutes for each student to talk while he did the timing explaining that he needed to do this because many of his class were reluctant to talk due to lack of confidence and familiarity with computers. He believed that this worked well with the start and stop guidance. The teacher linked this to Quality Assurance in big companies saying to the students that in the real world they need to communicate their work – this gave them purpose for the group work.

I am not very good at getting students that are reluctant to share coherently their experience when they do something new. I think I can do this better next time. I would like to coax them into this more to develop these skills.

Students kept the same groups throughout the program with the girls working together. Next time the teacher said he would do this a bit differently, intentionally mixing the students more. Teacher B was very positive about the extra help provided in the classroom by the university students.

One student wanted to extend ‘Hello Purr’ as a slideshow and they (student helpers) didn’t stop him and worked with him on the conditional logic. They provided good 1-1 support as well as being able to present up the front. It was good that the University girls were only slightly older, this helped the students relate to them. (Teacher B)

The intentional role-modelling was commented on by one other teacher, he stated:

[The female students] in particular were more hesitant in grabbing a phone. One had really low self-esteem – M [the Uni student] was great with her. I asked her if it was better for M to help or me...she replied M because she is a girl. The first girl coming into the group ended up doing more than some of the boys. She was hesitant initially. None of the girls wanted to be in the computer class and didn’t want to be with a room full of boys - “they are noisy and smell bad” [Teacher C]

While it is difficult to measure success in a small trial, one teacher reported that word of mouth seemed to occur in the playground and the recent course selections for 2013 indicated that student numbers for Year 10 IT next year would clearly increase, possibly three classes.

This is a very pleasing increase because there has been a spiral downward in numbers over the years”.

4.2 Student Helper Feedback

The student helpers kept weekly blogs on how each class went. Each of them was a volunteer who was studying an ICT degree program at university. They were given a copy of the text book and also the student curriculum as well as an android phone to practice on before going in to the classroom. They were rewarded at the end of the program with a book voucher and reference letter to thank them for their participation. In each case the student reflection provided another level of feedback to us on how well the students engaged with the curriculum.

The perception of Teacher A that the students were mainly viewers was not consistent with blog reflections:

Week 1: “I was helping students add sounds to their media pallet...I had to explain that all the coding happens within the blocks editor”

Week 2: “I spent most of my time helping one of the girls in the class”

Week 3: The questions became more complex which required E and I to search for the answer on the internet... There were some questions that were more about how to develop a function; for example adding a score function to their game.

The weekly reflections show the student’s learning, and also highlight the intentional role-modelling of normalising that girls can and do understand programming.

The student who worked with teacher B2 added insight as to why the implementation was less than successful:

Class 1: [the teacher] hadn’t brought any of the phones to the class for the students to use so they were using the emulator. This couldn’t test most of the extension work as a few features are unavailable as it’s just an emulator.

Class 2 [the teacher] still hadn’t figured out which task sheet the students were up to. [the teacher] had brought the phones to the class but had not unpacked the new ones.

Class 3 [the teacher] hadn’t photocopied the next task sheet for the students as most of them had finished the week 3 tasks.

The teacher and student helper feedback indicate that when the curricula was fully embraced by an enthusiastic teacher, the desired outcome of students having a compelling ICT experience was achieved. However not
all factors were within the power of the researchers or even the student facilitators.

5 Concluding Remarks
A student led peer support model curriculum was created, but we now ask “are teachers ready?”. It would appear that in this case two embraced the opportunity to allow their students to explore the program and share with each other, a third was quite entrenched in his own way of program delivery and on a tight time-frame so limited the implementation of the curricula and the fourth appeared to have other issues to contend with, such as classroom support and preparation time. It should be noted that the teacher B2 was a late replacement to the program and had not attended the initial teacher briefing workshop. We have no control over changes to staffing in schools, and while the initial teacher (B) was still involved in IT education in that school, it appears that teacher B2 had limited internal support.

The model for student led learning using artefacts (mobile phone) and drag and drop programming interfaces was generally positively embraced with students and our aim to build teacher technical efficacy and promote student – teacher learning partnerships was achieved in two of the four instance. We know that all the schools are continuing to run the program as part of their IT curriculum, and that one school has purchased its own class set of Android phones.

Our classroom facilitators were provided to encourage student to student interaction and take the pressure off the teacher somewhat. As can be seen by their comments they embraced this role. While one teacher observed that they were not utilised, the students reported that they were indeed helping individual students.

The use of the AppInventor tool to spark interest in programming appears to have achieved a positive outcome. This grant and outreach program has acted as a springboard to ongoing research opportunities. We have developed a workshop program for school teachers that is being delivered in an intensive mode to twenty teachers. The opportunity to share the findings from this first run of the program will alert them to some of the pitfalls experienced, e.g. the technical set up within their own school. The importance of teachers to take up the program and allow students to explore together in the classroom is integral to a successful outcome. Teachers need to embrace different pedagogies to allow students to explore as they learn.

A limitation of this paper is that it is based on only four implementations of the program. However it delivers a model, curriculum and structure that can provide greater school university interaction to promote the creativity and knowledge building of programming to middle school students via enthusiastic and competent teachers.

6 References


Appendix 1:
‘Create Your Own Apps’ – Teacher Notes and Unit plan - App Inventor, 4 week lesson plan (Written by Gail Casey, version 5th Sept 2013 – gccasey@deakin.edu.au)

Unit Overview – This is a flexible unit of work which takes a student centred approach to learning.

Teachers should use this document as a guide only. The Wolber App Inventor text is online at http://www.appinventor.org/projects

| Unit Title: ‘Create your own Apps’  
| Year Level: Year 7 to 11 - approximately 4 x 50 min  
| Unit Summary: |

This four week unit of work will help students understand the way in which mobile phones operate through the use of Apps. Students will research the online programming software ‘App Inventor’ before writing their own Apps using an Android mobile phone (or simulator). App Inventor fosters creativity through technology and is programmed through an Internet browser where students can design their own mobile phone apps. The program uses a series of blocks, like pieces of a puzzle, where students build a series of behaviours that when put together can appear live on an android phone.

Figure 1: What is App Inventor | Explore MIT App Inventor

http://explore.appinventor.mit.edu/content/what-app-inventor

Screen clipping taken: 15/07/2012, 8:06 AM
Things to note:
1. You need to be on the Internet to run App Inventor. It is a Web based application and runs by browsing to the App Inventor website, http://appinventor.mit.edu.
2. More resources for each of the tutorials can be found at http://www.appinventor.org/projects
3. The App Inventor Setup Installer software is something you need to install beforehand so that your computer can talk with the android phone or in the android emulator.
4. Many video tutorials have been created by David Wolber and links to his YouTube channel is given at http://www.appinventor.org/projects after clicking on the appropriate chapter, but for High school students, video tutorials made by Chris Groff should also be considered. Chris Groff's YouTube channel can be found at http://www.youtube.com/user/cgroff17
5. Some tasks involve students taking screen clips of websites to share with their peers. These could be done using the 'Prt Sc' key on the keyboard or using a screen capture software such as ‘Jing’ (See, http://www.techsmith.com/jing.html)
6. It is assumed that good Internet access is available.
7. Don’t use spaces when naming files

Possible areas for assessment could include:
1. Skills - ability of students to learn, design and create using App Inventor
2. Communication – ability to share one’s knowledge & communicate through the software.
3. Research & Investigation projects.

Please ensure that:
- Teacher and Students all have a Google account
- Java and the App Inventor software is loaded and runs on the computers
- You have backup activities for the classroom if the school does not have good internet access

Some suggestions in this unit plan:
- This four week unit is student-centred and works well if the teacher knows very little about App Inventor (or appears as such).
- Start lessons with a student-centred activity such as a student showcase, pair and share or a tips and hints session.
- Peer-to-peer learning is valued; hence, access to a shared space where program files and resources can be shared is helpful and supportive for collaboration. This could include an online drive, dropbox, wiki, blog or even school network drive.

Helpful Websites, other than http://ictplus.ning.com/:
- The main App Inventor site is at http://www.appinventor.mit.edu/ - click on ‘Invent’ to start programming.
- There are many helpful videos available on YouTube (See, http://www.youtube.com/watch?feature=player_embedded&v=VTbyqDCK3A0). Many of these can be downloaded prior to class for students to access from their school Intranet. Some are also available from Vimeo (See, https://vimeo.com/search?q=App+Inventor).
- Useful resource
  - Getting Started with Android App Inventor is http://www.i-programmer.info/programming/mobile/1789-getting-started-with-android-app-inventor.html, but be careful because it does refer to the old App Inventor site.
  - Many video tutorials have been created by David Wolber and links to his YouTube channel is given at http://www.appinventor.org/projects after clicking on the appropriate chapter, but for High school students, video tutorials made by Chris Groff should also be considered. Chris Groff's YouTube channel can be found at http://www.youtube.com/user/cgroff17
  - Other video tutorials also available at http://www.youtube.com/playlist?list=PLCF2969C390CE87F4
  - Other resources, http://www.appinventor.org/course-in-a-box
Create Your Own App: Programming with App Inventor Lesson Plan (4 Weeks)

**Week 1 – what is App Inventor & ‘Hello Purr’ (Task 1 & 2)**

**Introduce the topic** - the teacher instigates a class discussion by asking questions about mobile phones and mobile phone apps. This is to tease out what students know. The teacher should not have the answers and should not tell them what App Inventor is or where to find resources.

**Give students the Week 1 handout** - this is their instruction sheet and provides students with a check list for the first two tasks.

**Task 1 – What is App Inventor and why use it?**

This task encourages students to explore and find out more about App Inventor while sharing the resources they find with their peers. This task tries to personalise the experience by prompting students to think about what App Inventor can do for them.

The self and peer assessment is attempting to put the responsibility for learning onto the students. For example, if the work is not done then it is for the students, in their group meeting, to provide helpful feedback and direction to each other.

The teacher is responsible for identifying the ideal time for groups to meet each lesson. This involves students in groups of 3 around one computer and presenting their work to their group. If they don’t have access to a space where they can post their work, they could email the task to each group member. This allows them to easily click on the websites that each have found.

**Note:**
- Descriptions of ‘High’, ‘Medium’ and ‘Low’ for the peer and self assessment can be found at the end of the Week 1 Student handout.
- Peer assessment is an important part of Task 1 and sets the scene for the following tasks.
- It is advisable to collect the handout sheets at the end of each period so that they is not lost.
- Students have until the end of each week to use the advice from their peers as well as their own understandings to improve their work.

**Task 2 – Creating ‘Hello Purr’**

- Ensure that the pdf instructions and the ‘Hello Purr’ video tutorials are copied onto the school network so that students have access.
- Using the overhead projector show the students how to open both the App Inventor software and the video tutorial.
- Play the initial instructions from the first ‘Hello Purr’ video tutorial. Then minimise this and demonstrate the action using the App Inventor software. Continue to run the video and pause after each step to demonstrate within the actual software – you are modelling this method of learning. Encourage students to use this method on their own computers using headphones. Discourage them from watching the entire video and expecting themselves to remember all the steps.

By listing the skills gained after each tutorial the students are able to build the language of the software. For example, in the ‘Hello Purr’ skills include: renaming components, button properties, label properties, overlaying a button with a picture, adding sound, using blocks editor, connecting to Android phone.

**Task 2 – Sharing, Feedback and Assessment**

**Talented students should be encouraged to move on to Paint Pot while weaker students may strengthen their understanding by spending more time on modifying Hello Purr.**

**Possible Assessment for Week 1:**
1. Research
2. Task 1 & 2
3. Ability to help others.
Week 2 – Paint Pot Tutorial (Task 3)
Distribute the week 2 student handout. The next recommended tutorial is ‘PaintPot’. Where possible, the teacher should use a student to model the first part of one of the PaintPot tutorial (have both the video tutorial and the App Inventor software open and pause the video tutorial when working on the software).


Students continue with the tutorials and are supported by a group of peers.

Note:
1. Peer feedback & support – each period, get students together in their groups to discuss what they have done. They are expected to show their group the work that they have done and discuss any problems they have had.

2. Discussion of the PDF tutorial - the PDF versions of each tutorial can be very helpful. The groups should also be encouraged to look through the appropriate PaintPot pdf tutorial available. Using a combination of video and print media supports different learning styles. Also, after using the video tutorials, it is useful to view the pdf as they provide more detailed information.

Assessment:
1. Task 3 2. Ability to help others 3. Language of the software

Week 3 – Programming Jargon, Mole Mash Game and Review (Task 4 & 5)
During week 3 and 4, further theory from chapter 14 (Understanding an App’s Architecture) and 15 (Engineering and Debugging an App) may be useful to support and extend the programming concepts being used.

Distribute the week 3 student handout. Task 4 requires students to discuss a range of terms used – this should be done in their groups although each student should submit their own answers.

The recommended week 3 tutorial is ‘MoleMash’. There are five parts to the video tutorial for this game at http://www.appinventor.org/molemash. Note that at this site you can download the Chapter, download the source and download the APK file (package for the phone).

Where possible, the teacher should use a student to model part of a tutorial during each class – this could occur after students have been working on their programming for 15 min, when a talented student can be identified or when a number of students are having trouble.

Demonstrations should be optional for students to watch if they are at different stages.

During each class there should be some class discussion of tips and tricks as well as common problems

Students should meet in their group each lesson to provide peer feedback and constructive advice.

Assessment:
1. Task 4 & 5 2. Ability to help others.

Week 4 – Create your own App
This week students create an App of their choice or a, perhaps, a their own video help tutorial or handout with the aim of helping Year 7 students learn about App Inventor.

Distribute the Week 4 handout.
Students should meet in their group each lesson to provide feedback, support and peer assessment.

Assessment:
1. Task 6 2. Ability to help others (peers and Year 7).
It’s Not Them, It’s Us!
Why Computer Science Fails to Impress Many First Years

Rashina Hoda1 Peter Andreae2
1Electrical and Computer Engineering, The University of Auckland, New Zealand
2School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
r.hoda@auckland.ac.nz peter.andreae@ecs.vuw.ac.nz

Abstract
High attrition and failure in first year computer science and software engineering courses has often been linked to the personal traits and skills of students – dividing the world into those that “get it” and those “that don’t”. We present several concrete strategies based on the recently developed Learning Edge Momentum (LEM) theory, which when applied together, were found useful in reducing failure rates. Based on our experiences, we challenge our current understanding of attrition and failure in first year courses and dare to claim that maybe it’s not them, it’s us that is the problem.

Keywords: computer science, software engineering, first year course, attrition and failure rates, LEM theory

1 Introduction
Attrition and failure in first year computer science and software engineering courses has often been linked to the personal traits and skills of students, sometimes referred to as the “geek-gene”. According to this notion, the world can be divided into those that “get it” and those “that don’t”. In light of recent research emerging from the University of Otago, New Zealand (Robins, 2010), we attempt to redefine our current understanding of attrition and failure rates in first year courses.

The Learning Edge Momentum (LEM) theory challenges the notion of the “geek-gene” and suggests that it is the inherently interdependent nature of programming concepts, along with human tendency to learn at the edge of prior knowledge that is a significant contributing factor towards high attrition and failure rates (Robins, 2010). Fundamental concepts of programming imparted in first year courses are highly linked and “build upon” each other. This implies that an inability to grasp early concepts is a strong indicator of subsequent overall failure rates. We developed and introduced several strategies to our fundamental first year course based on the LEM theory. The results, although preliminary, are encouraging.

In this paper, we address one of the perennial problems of computer science – high failure and attrition rates in first year courses–and present some concrete strategies and encouraging results from our application of the LEM theory.

2 Background
COMP102: Introduction to Program Design is a first course in programming in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand. The course introduces object-oriented programming, with objects introduced fairly early in the course. The course spans one trimester (12 weeks) and introduces Java control structures, methods, parameters, top-down design, text input/output, graphical output, objects and classes, files, arrays (1D, 2D, variable sized lists), simple event-driven GUI (very constrained), and Java interfaces. We do not cover inheritance or Collection classes in this course. COMP102 is a mandatory course for all computer science and engineering majors and a popular elective for some other disciplines, such as Information Systems. In other words, COMP102 is a reasonably standard first year Computer Science/Software Engineering course. Further details of the course structure and content can be found on the course [homepage](http://ecs.victoria.ac.nz/Courses/COMP102_2011T1/)

3 The Problem: High Attrition and Failure Rates
Over the past 25 years, through all its minor and major modifications and variations, COMP102 has consistently exhibited high attrition and failure rates, ranging from 40 to 50%. This is a problem. Such high failure rates are common in similar courses around the world and so are the non-normal distributions of grades (especially a bimodal distribution).

Although research into computer science education does not conclusively identify anyone or more factors that determine success or failure (Bornat, Dehnadi and Simon 2008, Cross 1970, Curtis 1984), several factors have been suggested as possible causes of high attrition and failure rates in first year programming courses. One of the most common is the notion that individuals have an innate ability to program which determines their success or failure. Determinants of this “innate programming ability”, suggested over the years, include factors such as cognitive ability (verbal, mathematical, spatial, and analogical skills) (Pea and Kurland 1984, Wileman, Konvalina and Stephens 1981, Wolfe 1969) cognitive development (Piaget’s stages of cognitive development and Bloom’s taxonomy of educational objectives) (Bloom, Englehart, Furst, Hill and Krathwohl 1956, Piaget 1971), cognitive style (learning style, personality type etc.) (Hudak, and Anderson 1990, Myers 1995), and demographic factors (gender, age, etc.) (Wosczynski, Haddad and Zgambo 2005). In other words, most
research has been focused on determining the cause assuming the problem is with “them” (i.e. the students)

4 The LEM Theory

The Learning Edge Momentum (LEM) hypothesis suggests an alternative explanation and claims that it is the human tendency to learn at the edge of prior knowledge combined with the inherently tight and highly interdependent nature of programming concepts that leads to success or failure in learning programming (Robins, 2010). In other words, since we learn at the edge of what we already know, successful acquisition of one concept makes it easier to learn other closely related concepts and vice-versa (Robins, 2010). At the heart of the LEM theory is the realization that the nature of programming is such that concepts (and constructs) “build upon” each other and failure to grasp any one component, especially in the early parts of the course, has a cascading effect – making it significantly harder to grasp later, related concepts. The LEM hypothesis is based on a simulated model of grade distributions and an extensive review of educational and psychological literature.

Our experiences suggest that programming inevitably involves dependence e.g., one cannot understand loops without understanding variables, and one cannot understand arrays without understanding loops, and so on. This highly-integrated nature of programming concepts coupled with the way people learn creates an “inherent structural bias” in first year courses leading to extreme outcomes reflected by bi-modal distribution of grades.

5 The Strategies: Improving Momentum

A direct recommendation of the LEM theory is for particular attention to be paid to early stages of the course ensuring everything runs smoothly and there are plenty of opportunities for grasping early concepts. Robins’ recommendations, however, were very general. To apply these recommendations, and the principles of the LEM theory, we developed a set of concrete strategies for modifying COMP102. They can be grouped into four clusters below and described in the following subsections:

- Minimizing early complexities in the course
- Minimizing dependences between early components of the course
- Maximizing chances of mastery of the early concepts and skills
- Maximizing opportunities for early recovery

In the following sections, we describe each of these.

5.1 Minimizing Early Complexity using UI Library

An ideal course from a LEM perspective would start with modules that each address a small set of concepts, skills, and knowledge, and able to be learned readily by students based on what they already knew at the beginning of the course. A typical programming course, especially in a language such as Java, has a large number of “gratuitous complexities” – concepts that are not fundamental principles of programming but are consequences of the programming language, the programming environment, or the particular details of how the lecturer has chosen to present the material.

Fig. 1 Example of using UI library (right) to minimize early complexity

Even simple one-method programs in Java involve a lot of gratuitous complexity if they involve any input and output. For example, standard output using System.out.println involves calling a method on a static field. Even though this does not have to be explained in detail, this statement has two “dots”, in contrast to the standard pattern of <object> <dot> <method name> ( <arguments>) and such inconsistencies constitute gratuitous complexity that trips up students. Standard input also includes similar complexities. The simplest form is probably to use a Scanner, but this means that for their first programs to have any input from the user, the students must deal with creating instances of a Class (and passing an argument that is a static field to the constructor), storing the object in a variable, and then calling methods on it. To use any kind of graphical output requires even more complexity. Although experience tells us that many students cope with this, the LEM theory also suggests that some students will fail the course because they got tripped up at this early stage and were unable to recover.

We designed and introduced a Java library (the “UI” library) that provides much simpler input and output, allowing students to do text input and output, and simple graphical output by calling methods on a “predefined object”. This library removed a lot of the gratuitous complexity from the early part of the course. Importantly, it made it possible to delay the construction of new objects from the second week to the third week, significantly simplifying the concepts required for the second assignment. It also allowed the construction of new objects to be introduced in a more meaningful and motivating context, rather than just as a way of getting input from the user. The library was designed to be as consistent with standard Java as possible, in order to minimize the barriers when the students have to deal with standard Java. For example, text input in the library includes methods with the same names and behaviour as the methods in the Scanner class, making it easier for students to cope with Scanner when they meet it in the context of reading data from files later in the course.

Many courses and textbooks have also introduced special libraries to reduce the complexity for new programmers. However, the UI library seems to be particularly simple to use, in comparison to the ones we
have seen. More details: http://ecs.victoria.ac.nz/Courses/COMP102_2011T1/Comp102Documentation

Fig. 1 shows a couple of examples of how the use of the UI library minimized complexity with the original code on the left-hand side and the same code simplified as a result of the use of the UI library on the right-hand side.

5.2 Minimizing Dependencies in Assignments

The second ideal quality from a LEM perspective is that the modules should not depend on each other, so that students can learn each module based on what they already knew at the beginning of the course, rather than having to have already mastered the previous modules. As Robbins points out, the ideal is simply not possible in programming since so many of the concepts build on top of each other – for example, parameter passing depends on understanding variables, and both conditionals and loops depend on Boolean expressions. However, since many of the early assignments had to be at least modified, if not replaced because of the new library, we were able to look again at the assignments from the perspective of minimizing dependencies. By being careful about choosing the programming tasks, we were able to significantly reduce the level of dependence between assignments 2, 3 and 4, compared to the previous year. For example, there were two pairs of programs prior to introducing LEM strategies where the second program in the pair was an extension of the first program assigned in the previous week. If a student failed in the earlier assignment, they were at an immediate and obvious disadvantage in the later assignment. In introducing LEM strategies, we eliminated all such pairs, so that each program in the first four weeks was quite different.

The changes to the library also removed some of the dependencies, so that there was no longer a dependency between the program that introduced text input and the program that was centred on creating new objects and calling methods on them (since dealing text input no longer had to introduce the concept of creating a new Scanner object).

However, there was little reduction in the dependencies between the later assignments, because they were deliberately addressing larger programs that necessarily integrated a variety of constructs and concepts from the earlier part of the course.

5.3 Maximizing Chance of Success using “Bridging Exercises”

Even though we were able to reduce some of the gratuitous dependencies between the early assignments, there were still significant dependencies, even in the first four weeks. For example, variables are introduced right at the beginning, and are used in all programs from then on; once conditionals are introduced, they are used everywhere. We believe that these dependencies are unavoidable.

Given this, it is essential to maximize the probability that students will be able to master the concepts in every one of the early assignments. This is not necessarily the same as maximizing the probability of successfully completing all the programs – all that is required to keep the momentum going is for the students to understand the new concepts in each module well enough to be able to use them and build on them in the next module.

Our previous assignments were all whole programs, and if they didn’t get the program, they probably didn’t get the concept either. We did not want to get rid of these “whole programs” – represent what the larger task of programming is all about and the fundamental goal of the course – but the “all or nothing” aspect is problematic according to LEM theory.

Therefore we added exercises – to enable mastery of the individual constructs and concepts, as a “bridge” into the programs which would then build on and solidify, and show their use in a realistic context. The goal of the exercises was merely mastery of individual new constructs and new concepts. Exercises were small artificial programs that were pared down to be as small as possible without being totally meaningless. Students were allowed to get as much help from tutors in the labs as they needed for completing the exercises. In order to avoid the exercises becoming a possible hindrance for the more advanced students, there were a series of exercises which were not marked and students could move to the actual (marked) program as soon as they could do 2 exercises by themselves.

5.4 Maximizing Opportunities for Early Recovery via Self-Directed e-Learning

We developed several self-directed e-learning tools to allow students maximum opportunities for revisiting materials and learning from them in a self-paced manner. These self-directed tools included video materials that were made available online to students in order to provide them with the ability to self-direct their learning. The lectures were video recorded and the recordings were made available to students online through the course homepage and in their labs. Lecture videos allowed students to view/review material in their own time at their own pace. We produced several kinds of videos: video recordings of lectures, demos of assignment programs, working review of past tests and exams, short tutorials on various topics, and additional review of lecture material. We also provided videos demonstrating the assignment programs to make sure that students understood clearly what was required.

These materials included a set of short, “YouTube-style” tutorial videos focused on single programming concepts, such as loops and methods, and working through previous exam questions. These videos were between 8 and 30 minutes. Tutorial videos took double the time to prepare as the length of the videos but were reusable from year-to-year.

5.5 Encouraging Results

We are encouraged by the preliminary results of applying the LEM theory to COMP102. Preliminary results show that the overall failure reduced to 35% (from 45% the previous year). A comparison between failure rates in before and after application of LEM theory is presented
in Table 1. There were 262 and 269 students in the course in each of the years respectively.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Pre-LEM</th>
<th>Post-LEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall (of 262/269)</td>
<td>45%</td>
<td>35%</td>
</tr>
<tr>
<td>CS/ENG (of 173/169)</td>
<td>39%</td>
<td>33%</td>
</tr>
<tr>
<td>Non-CS/ENG (of 89/100)</td>
<td>52%</td>
<td>37%</td>
</tr>
<tr>
<td>Design Students (of 17/19)</td>
<td>75%</td>
<td>42%</td>
</tr>
<tr>
<td>No prior programming experience (of 127/149)</td>
<td>48%</td>
<td>42%</td>
</tr>
</tbody>
</table>

Table 1. Failure Rates in COMP101 Pre- and Post-Application of LEM Theory Strategies.

We conducted a course evaluation at the end of the course to gain a sense of how our strategies were perceived by the students. There were 128 responses, of which 68% indicated that they found that the exercises and lecture videos "contributed to learning"; nearly 72% said they found that the tutorial and demo videos "contributed to learning".

We also analysed the written comments on evaluation forms which favoured video resources due to their ability to help students in "revisiting concepts", "catching missed lectures", "seeing assignments work before starting on it", and easily accessing them. Similar comments were recorded for tutorial videos: "Tutorial videos helped a lot - need more of them", "tutorial videos going over last year's test helped".

We believe that if these preliminary results hold up, then there is merit in continuing with the strategies we developed and deployed in COMP102 based on the LEM theory. Further iterations of the course will provide a better indication of the sustainability of these results.

6 Conclusion

High attrition and failure rates in first year computer science and software engineering courses have traditionally been attributed to individuals' "innate" inability to program. Recent research proposed an alternative explanation in the form of the Learning Edge Momentum (LEM) theory which suggests that human tendency to learn at the edge of prior knowledge combined with the inherently tight and highly interdependent nature of programming concepts leads to success or failure in learning programming. We developed some concrete strategies in order to apply the LEM theory to our first year computer science and software engineering course and found encouraging results.

Using the strategies presented in this article – such as reducing dependencies between components and providing ample avenues for successfully grasping core concepts early on – we hope to provide everyone who attempts to learn programming a better chance at succeeding.

7 References


<table>
<thead>
<tr>
<th>Author</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andreae, Peter</td>
<td>159</td>
</tr>
<tr>
<td>Bittner, Klaus</td>
<td>97</td>
</tr>
<tr>
<td>Carbone, Angela</td>
<td>123</td>
</tr>
<tr>
<td>Casey, Gail</td>
<td>149</td>
</tr>
<tr>
<td>Cooper, Graham</td>
<td>139</td>
</tr>
<tr>
<td>Cornforth, David</td>
<td>75</td>
</tr>
<tr>
<td>Criag, Annemieke</td>
<td>149</td>
</tr>
<tr>
<td>D’Souza, Daryl</td>
<td>iii, vii, 113</td>
</tr>
<tr>
<td>de Salas, Kristy</td>
<td>21</td>
</tr>
<tr>
<td>Dermoudy, Julian</td>
<td>21, 113</td>
</tr>
<tr>
<td>Dickerhof, Markus</td>
<td>97</td>
</tr>
<tr>
<td>Ellis, Leonie</td>
<td>21</td>
</tr>
<tr>
<td>Falkner, Katrina</td>
<td>3</td>
</tr>
<tr>
<td>Falkner, Nickolas</td>
<td>3</td>
</tr>
<tr>
<td>Heinsen, Egan Matthew</td>
<td>51</td>
</tr>
<tr>
<td>Herbert, Nicole</td>
<td>21</td>
</tr>
<tr>
<td>Hoda, Rashina</td>
<td>159</td>
</tr>
<tr>
<td>Hu, Minjie</td>
<td>113</td>
</tr>
<tr>
<td>Jones, Asheley</td>
<td>13</td>
</tr>
<tr>
<td>Kasto, Nadia</td>
<td>105, 133</td>
</tr>
<tr>
<td>Lang, Catherine</td>
<td>149</td>
</tr>
<tr>
<td>Lewis, Ian</td>
<td>21</td>
</tr>
<tr>
<td>Lister, Raymond</td>
<td>41, 65</td>
</tr>
<tr>
<td>Lopez, Dobrila</td>
<td>31</td>
</tr>
<tr>
<td>Lopez, Mike</td>
<td>31</td>
</tr>
<tr>
<td>Luxton-Reilly, Andrew</td>
<td>59</td>
</tr>
<tr>
<td>MacDonald, Chris</td>
<td>91</td>
</tr>
<tr>
<td>Mason, Raina</td>
<td>139</td>
</tr>
<tr>
<td>McDonald, Chris</td>
<td>51</td>
</tr>
<tr>
<td>Miller, Charlynn</td>
<td>13</td>
</tr>
<tr>
<td>Mustafa, Umniya</td>
<td>83</td>
</tr>
<tr>
<td>Parsons, Dale</td>
<td>113</td>
</tr>
<tr>
<td>Philpott, Anne</td>
<td>133</td>
</tr>
<tr>
<td>Putro, Iwan Handoyo</td>
<td>123</td>
</tr>
<tr>
<td>Schmidt, Andreas; Kimmig, Daniel</td>
<td>97</td>
</tr>
<tr>
<td>Sheard, Judy</td>
<td>113, 123</td>
</tr>
<tr>
<td>Simon, 75</td>
<td>113</td>
</tr>
<tr>
<td>Skudder, Ben</td>
<td>59</td>
</tr>
<tr>
<td>Teague, Donna</td>
<td>41, 65</td>
</tr>
<tr>
<td>Trabelsi, Zouheir</td>
<td>83</td>
</tr>
<tr>
<td>Vivian, Rebecca</td>
<td>3</td>
</tr>
<tr>
<td>von Konsky, Brian</td>
<td>13</td>
</tr>
<tr>
<td>Whalley, David</td>
<td>133</td>
</tr>
<tr>
<td>Whalley, Jacqueline</td>
<td>iii, vii, 105, 133</td>
</tr>
</tbody>
</table>
Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series Conferences in Research and Practice in Information Technology. The full text of most papers (in either PDF or Postscript format) is available at the series website http://crpit.com.

Volume 124 - Database Technologies 2012

Volume 125 - Information Security 2012

Volume 126 - User Interfaces 2012
Edited by Baiqing Chen, Flinders University, Australia and Ruz T. Smith, University of South Australia, Australia. January 2012. 978-1-921770-07-4.

Volume 127 - Parallel and Distributed Computing 2012
Edited by Jujuan Chen, University of Technology, Sydney, Australia and Rajan Ranjan, CSIRO ICT Centre, Australia. January 2012. 978-1-921770-08-1.

Volume 128 - Theory of Computing 2012

Volume 129 - Health Informatics and Knowledge Management 2012

Volume 130 - Conceptual Modelling 2012

Volume 133 - Australian System Safety Conference 2011

Volume 134 - Data Mining and Analytics 2012
Edited by Zhi Zeng, Department of Immigration and Citizenship, Australia, Jiuyong Li, University of South Australia, Australia and Peter Christen, Australian National University, Australia. December 2012. 978-1-921770-14-2.

Volume 135 - Computer Science 2013
Edited by Bruce Thomas, University of South Australia, Australia. January 2013. 978-1-921770-20-3.

Volume 136 - Computing Education 2013
Edited by Angela Curbach, Monash University, Australia and Jacqueline Whalley, AUT University, New Zealand. January 2013. 978-1-921770-21-0.

Volume 137 - Database Technologies 2013

Volume 138 - Information Security 2013

Volume 139 - User Interfaces 2013
Edited by Ruz T. Smith, University of South Australia, Australia and Burkhard C. Wünche, University of Auckland, New Zealand. January 2013. 978-1-921770-24-1.

Volume 140 - Parallel and Distributed Computing 2013

Volume 141 - Theory of Computing 2013

Volume 142 - Health Informatics and Knowledge Management 2013

Volume 143 - Conceptual Modelling 2013

Volume 144 - The Web 2013
Edited by Helen Ashman, University of South Australia, Australia, Quan Z. Sheng, University of Adelaide, Australia and Andrew Trowbridge, University of Otago, New Zealand. January 2013. 978-1-921770-15-9.