Psychology of Programming Interest Group (PPIG) Work-in-Progress Workshop 2013

School of Computing and Mathematics
Keele University, UK
8 – 9 July 2013

Proceedings
Edited by Louis Major
Welcome to the Psychology of Programming Interest Group (PPIG) Work-in-Progress Workshop 2013, taking place in the School of Computing and Mathematics at Keele University (Staffordshire, UK).

The Psychology of Programming Interest Group (PPIG) was established in 1987 in order to bring together people from diverse communities to explore common interests in the psychological aspects of programming and/or in computational aspects of psychology. The group attracts cognitive scientists, psychologists, computer scientists, software engineers, software developers, HCI people et al., in both universities and industry.

As always at PPIG, this year's workshop involves discussion on a range of topics. Presenters have travelled from countries including Australia, Canada, Germany, the United States and the United Kingdom.

On the second day of the workshop we welcome Gordon Rugg as a Guest Speaker. Gordon is a former timberyard worker, archaeologist and English lecturer. His PhD was in experimental social psychology. He subsequently worked in knowledge elicitation for expert systems, then in expert systems for information retrieval, after which he moved into requirements engineering and HCI/human factors. His recent work has mainly involved development of a framework for identifying errors in expert reasoning about long-standing problems. Gordon is a senior lecturer in computing at Keele University, and a company director.

We are grateful to the members of the technical committee, organising committee and the reviewers for their hard work preparing for this workshop. Finally, we would like to thank the authors and speakers, without whom such a high quality technical program would not be possible.

Louis Major
## Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Author(s)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Anxiety and the potential for mitigation</td>
<td>Sarah Crabbe and Peter Andras</td>
<td>3</td>
</tr>
<tr>
<td>Neo-Piagetian Preoperational Reasoning in a Novice Programmer</td>
<td>Donna Teague and Raymond Lister</td>
<td>10</td>
</tr>
<tr>
<td>Increasing Student Confidence throughout the Computer Science Curriculum</td>
<td>John A. Trono</td>
<td>12</td>
</tr>
<tr>
<td>The cognitive difficulties of Description Logics</td>
<td>Paul Warren, Trevor Collins and Paul Mulholland</td>
<td>18</td>
</tr>
<tr>
<td>Towards Defining a Cognitive Linguistics of Programming and Using Eye Tracking to Verify Its Claims</td>
<td>Sebastian Lohmeier</td>
<td>24</td>
</tr>
<tr>
<td>Representational Formality in Computational Thinking</td>
<td>Alistair Stead and Alan Blackwell</td>
<td>27</td>
</tr>
<tr>
<td>Towards a Mechanism for Prompting Undergraduate Students to Consider Motivational Theories when Designing Games for Health</td>
<td>Monica M. McGill</td>
<td>30</td>
</tr>
<tr>
<td>Analysing the effect of Natural Language Descriptions as a support for novice programmers</td>
<td>Edgar Cambranes, Judith Good and Benedict du Boulay</td>
<td>34</td>
</tr>
<tr>
<td>Workshop: Psychological Impact of 3D Intelligent Environments on Pedagogy</td>
<td>Martin Loomes, Georgios Dafoulas and Noha Saleeb</td>
<td>36</td>
</tr>
<tr>
<td>Bringing Heterogeneous Computing to the End User</td>
<td>Gordon Inggs &amp; David B. Thomas and Wayne Luk</td>
<td>39</td>
</tr>
<tr>
<td>Integrating Connectivist Pedagogy with Context Aware Interactions for the Interlinked Learning of Debugging and Programming</td>
<td>Xiaokun Zhang</td>
<td>41</td>
</tr>
</tbody>
</table>
Computer Anxiety and the potential for mitigation

Sarah Crabbe
York St John Business School
York St John University
s.crabbe@yorksj.ac.uk

Peter Andras
School of Computing Science
Newcastle University
peter.andras@newcastle.ac.uk

Abstract
This paper sets out the findings and further research about potential links between personality, computer anxiety and the mitigation strategies that can be used to alleviate the level of anxiety felt by an individual. Having found that, while there is a correlation between openness and emotional stability and computer anxiety this is not enough to successfully predict those who would be likely to suffer. Given that a significant minority of people will suffer from computer anxiety at some point and that this is often hidden or ignored, potentially leading to reduced performance or even stress related illness, mitigation of this would seem to be important. The research is investigating the various ways that people like to be supported and seeing if there is a link between this and their personality type.

Index Terms—computer anxiety, technophobia, personality

Introduction
The use of computers is an everyday occurrence for the majority of people at work, home and in education. While for many this is an enjoyable experience for a sizeable minority this interaction causes anxiety. How this anxiety manifests and can be mitigated efficiently and effectively is worthy of investigation. Prior research (Crabbe & Andras, 2012) indicated that there was a weak link between low scores in both emotional stability and agreeableness and a high level of computer anxiety. However this was not enough to confidently predict an individual’s level of computer anxiety. Building on this we set out a research strategy to identify the level of computer anxiety and identify the range of support strategies to deal with this. The possibility that the preference for strategy is linked to personality type is also explored.

What knowledge is needed?
There are several aspects of a person that need to be assessed in order to arrive at a suggestion for mitigation and then to verify that this was the correct intervention.

- Personality
- Level of anxiety
- Support strategy preference
- Success of preferred strategy

Personality
While there are several different ways of describing personality, most researchers are agreed that personality does not change very much over time (Maltby, Day, & Macaskill, 2007; Nettle, 2007) and there are a range of tools designed to measure the different factors that make a person who they are. Many of these are self-reporting questionnaires which expect the person to have some emotional intelligence i.e. can use emotive vocabulary to describe themselves.

For many questionnaires there are large bodies of research that justify the approach taken (Bradley & Russell, 1997; Ceyhan, 2006; Choi, Deek, & Im, 2008; Durndell & Haag, 2002; Kaiser, McGinnis, &

Many of these questionnaires can be administered by a researcher or even at a distance using an electronic survey tool.

Often the results of the test are numeric and based on scales. The individual is not described as this or that, but as being strongly or weakly this or that. This gives rise to a profile that is unique for each individual and is not so likely to classify people as one type or another. This is part of their attraction alongside the relatively low cost or open access to the tools and the research that they support.

There are other tools like MBTI (Briggs Myers, 2000) which have to be administered by trained psychologists. There is a cost in purchasing the tests and the mark schemes. Although the results are often interesting and the test has been around for a long time there are beginning to be some questions about its validity (Boyle, 1995; “Myers-Briggs Widely Used But Still Controversial,” n.d.) The usefulness in classifying people into one of only 16 pigeon holes is also quite debatable given the complexity of humans.

The group of factors called the Big Five (Nettle, 2007) are more accessible and there are open-source, well researched questionnaires based on this model that are available for general use. These can be administered by a researcher without psychological training and are completed by the subject at a time of their own convenience as they do not require the presence of the researcher. The completed results can be quickly marked and the response given back to the subject.

Level of computer anxiety

The level of computer anxiety can be measured in a variety of ways. Among the most popular seem to be:

The Computer Anxiety Rating Scale (Rosen & Weil, 1992) which has been used in many research projects (Abd-el-fattah, 2005; Anderson, 1996; Anthony, Clarke, & Anderson, 2000; M. Brosnan, Gallop, Iftikhar, & Keogh, 2010; M J Brosnan, 1999; Heinssen, 1987; Hvelka, Beasley, & Broome, 2004; Karal, 2009; Korobili et al., 2010; Maurer, 1994; Meier, 1988; Rahimi & Yadollahi, 2011; Rosen & Weil, 1995a; Tekinarslan, 2008). This 20 item questionnaire with 5 point Likert Scales results in a single rating which indicates the level of anxiety suffered,

The Computer Attitude Scale (Nickell & Pinto, 1987). This was analyzed by Ceyhan and Nalum to suggest three factors, and has been used in a range of research projects too. (Agyei & Voogt, 2011; Altinkurt & Yilmaz, 2012; Celik & Yesilyurt, 2013; Karal, 2009; Kurt & Gürcan, 2010; Rahimi & Yadollahi, 2011; Tekinarslan, 2008).

The Computer Anxiety Scale developed by George A Marcoulides in 1989 which seems to be among the most popular scale in use at the moment (Powell, 2013) and has been revalidated many times. This uses a two pronged approach with equipment anxiety being separated from general computer anxiety. (John J. Beckers, Wicherts, & Schmidt, 2007; M J Brosnan & Thorpe, 2006; M. J. Brosnan, 1998; Chua, Chen, & Wong, 1999; Hewson & Brosnan, 2002; López-Bonilla & López-Bonilla, 2012; Marcoulides, Emrich, & Marcoulides, 2011; Morris, Hall, Davis, Davis, & Walton, 2003; Namlu, 2003; Rosen & Weil, 1995b; Schulenberg & Melton, 2008; Terzis & Economides, 2011; Torkzadeh, Chang, & Demirhan, 2006; van Raaij & Schepers, 2008; Weil & Rosen, 1995; Wicherts & Zand Scholten, 2010)

The six-factor model (J J Beckers & Schmidt, 2001) recognizes that computer anxiety is complex and influenced by the experiences of the user and how these are perceived and internalized. An identical experience can cause quite different reactions in different people and give rise to different levels of computer anxiety. This has been used to support work in a range of studies (Hvelka et al., 2004; Korobili et al., 2010; Teo, 2008).
Support Strategy Preference

People are complex and there are many reasons for computer anxiety which may have an impact on how support should be given which can be organized into four different areas.

1. Not knowing how to start the application for example gives rise to operational anxiety
2. Conditioning such as being told that computers are difficult to use is a sociological cause
3. A deep mistrust of technology can give rise to a more psychological anxiety
4. Bad prior experience such as being taught badly can cause experiential anxiety

Three of the types of anxiety identified above: Operational, sociological and psychological were noted as long ago as 1986 (Howard, 1986).

Operational computer anxiety is potentially the easiest to mitigate as it is caused by a lack of skills knowledge. This can be addressed with appropriate training. (Anthony et al., 2000; Blazhenkova & Kozhevnikov, 2010; Cowan & Jack, 2011; Gravill & Compeau, 2008; Gupta, Bostrom, & Anson, 2010; Heinssen, 1987; van Raaij & Schepers, 2008)

Helping the user to know what to do can mitigate some of the anxiety that they feel. How this help is actually delivered is also an interesting issue as people have different learning preferences. As an example, some may prefer to have a manual, while others prefer to be taught by an expert. This could be discovered by asking the user what their preference is or it may be linked to the personality type of the user.

Sociological computer anxiety is a bit more difficult to address as it can include cultural issues, gender stereotypes and peer pressure. This could perhaps be mitigated through the use of coaching, positive role-models and more general education about the purposes and uses of the specific application increasing the desire to acquire the skill.

Psychological computer anxiety may only be successfully addressed through professional counseling although perhaps the impacts can be reduced using similar methods to those suggested above.

The fourth, experiential anxiety was noted more recently (Cowan & Jack, 2011). This is also difficult to completely mitigate as an experience has happened and cannot be undone. However it may be possible to reframe the response of the affected party using coaching techniques, or to reduce the impact of a bad experience by overlaying it with many positive experiences.

There is also much discussion about self-efficacy (Bandura, 1994) and its impact on the level of computer anxiety (John J. Beckers et al., 2007; M. J. Brosnan, 1998; Celik & Yesilyurt, 2013; Chou, 2001; Durndell & Haag, 2002; Karsten, Mitra, & Schmidt, 2012; Lindblom, Gregory, Wilson, Flight, & Zajac, n.d.; Mcilroy, Sadler, & Boojawon, 2007; Torkzadeh & Van Dyke, 2002). Self-efficacy is tied up with people’s idea of their own skill level as well as their actual skill level. It would seem to be sensible to assume that increasing self-efficacy would reduce computer anxiety but it seems more likely that reducing computer anxiety increases self-efficacy (M. J. Brosnan, 1998)

People have their own ways of learning and if this is not addressed effectively it can have a negative impact on performance and may feed into increased anxiety levels. (Altinkurt & Yılmaz, 2012; Austin, Nolan, & Donnell, 2009; Beckwith & Burnett, 2004; Bennett, Maton, & Kervin, 2008; Chien, 2008; Furnham, Monsen, & Ahmetoglu, 2009; Kurt & Gürcan, 2010; Margaryan, Littlejohn, & Vojt, 2011; Tulbure, 2011) While it can be seen that training is most successful when it matches the learning style of the user (Gupta et al., 2010)

One way of finding out how people want to be supported in their learning is to ask them. A brief questionnaire can ascertain what support has been used before and which support the user thinks would work for them best in the future.

Success of preferred strategy

How successful the intervention has been could be ascertained by repeating the measurement of the initial computer anxiety level and seeing if this has altered.
Plan for data collection

Three different cohorts will be assessed in September

1. York St John University Business School Level 1 Business Management Undergraduates
2. York University Social Science Level 1 Undergraduates
3. Robert Kennedy College Masters Level Students across several programmes (MALIC, MBA, MBA Sustainability etc.)

Cohorts 1 and 2 will be predominantly under 25 years old, first degree students based on campus in the UK. Cohort 3 will be predominantly over 35 years old, accessing a Masters course on-line as distance learners and will be based around the world.

The questionnaires about the three areas: computer anxiety, personality and support preference, will be collated into one questionnaire in order to collect a comprehensive picture of each respondent. The results should clarify which are the most popular methods of support to enable the development and application of appropriate support strategies.

The results may indicate whether on campus students have different needs from distance learners and different levels of computer anxiety. Distance Learners have often chosen this way of learning because other options are limited, but anecdotally it seems that they would prefer to learn in a more social environment.

It would be interesting to see if different academic disciplines present with differing levels of anxiety or have different requirements for support, and if there is any personality differential between the cohorts.

Students with high levels of computer anxiety in Cohort A will be given the option to access a range of support strategies that have been identified as to being appropriate for them.

At the end of the first semester all the participants will be re-tested using the first computer anxiety measure to see if their anxiety level has changed.

Analysis

The questionnaires will return two sets of quantitative data and a mixed set of qualitative and qualitative. This data pool can be analysed for correlations between results. The level of change of computer anxiety will be checked for the whole group and the difference analysed to see if the students who had interventions reduced their anxiety levels.

As there are three sets of students there it will be interesting to compare the scores across the cohorts and see if there are any correlations between age groups, gender, levels of experience, nationality and programme being studied.

Conclusion

The project is an exploration. There is no hypothesis to be proven or challenged. It is a moment to collect data and see if anything interesting emerges. The opportunity to access a large and varied population is exceptional and the information gleaned may help all students to achieve better results.

The expectation is that this may lead to a novel approach to predicting the most appropriate intervention strategy for mitigating computer anxiety.

Acknowledgment

SC acknowledges the support of the staff at YSJU Business School especially Doug who will help to implement the project, to Matt Cox at YU for sharing the questionnaires with the students there and for RKC for allowing access to their students. Also the students who are going to give their time to take part in this work.
References


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, due to both the well known novice misconceptions about language semantics and their inability to organize a written manual execution of the code.

- **Preoperational**: The novice can manually execute 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 manually execute the code, perhaps more than once with differing initial values, and make an educated guess based on the input/output behaviour.

- **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 executing the code with specific values. When reading code, they tend to reason in terms of constraints on the set of possible values that each variable may have. The concrete operational stage is the first stage at which novices can reason with abstractions of code, such as diagrams.

- **Formal Operational**: As this most advanced stage does not figure in this paper, we refer the reader to Lister (2011) for a description of it.

It is well known that researchers since Piaget have conducted experiments that call into question aspects of his “classical” Piagetian theory. Neo-Piagetians have proposed modifications to address problematic aspects of the classical theory. One of the problematic aspects is the conception of stages. For example, in classical Piagetian theory, a person spends an extended period in one stage, before undergoing a rapid change to the next stage. This is commonly referred to as the “stair case metaphor”. Instead, some neo-Piagetian researchers have found evidence for the “overlapping wave” metaphor (e.g. Siegler, 1996), where a person exhibits a changing mix of reasoning strategies from the different Piagetian stages. Initially, the sensorimotor stage of reasoning is dominant, but its frequency of use declines as the use of preoperational styles of reasoning increases, and so on. Our think aloud data supports the overlapping wave model.

Corney et al. (2012) provided indirect evidence that novice programmers pass through the preoperational and concrete operational stages, by analyzing student answers to exam questions. 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 the study reported here, we had several volunteer students complete programming related tasks while “thinking aloud”. We found direct evidence that students progress though these neo-Piagetian stages. We met approximately once each week with these students, so we could follow their progress over the course of a semester. In this abstract, we briefly describe a single think aloud session with
one representative student, with the pseudonym "Donald". He is 22 years old, speaks English as his first language and at the time of the think aloud described below he was in his 9th week of semester. In one exercise, Donald (and other participants) were asked to read the mix of python code and pseudocode shown below and then, in one sentence, “describe the purpose of that code”:

```python
if (y1 < y2):
    Code to swap the values in y1 and y2 goes here.
if (y2 < y3):
    Code to swap the values in y2 and y3 goes here.
if (y1 < y2):
    Code to swap the values in y1 and y2 goes here.
```

Donald (and some other participants) used the inductive approach of a novice at the preoperational stage. That is, Donald selected some initial values for the variables (y1 = 1, y2 = 2 and y3 = 3), manually executed the code with those values, and then inferred what the code did from the input/output behaviour. In his think aloud, Donald initially showed remnant attributes of the sensorimotor stage, as his first attempt at manual execution was unorthodox and poorly organised. After he gave up on that initial attempt, he then spontaneously adopted a more orthodox and better organised approach to manual execution, and quickly completed a correct manual execution. However, he then made an incorrect inductive inference, which led him to write the following incorrect answer: “To reverse the values stored in y1, y2 and y3 ...” The person conducting the think aloud (the first author) then provided Donald with alternate initial values for the variables (y1 = 2, y2 = 1 and y3 = 3). Donald then performed a successful manual execution with those values. On completing that execution, Donald exclaimed, “Oh! It’s ordering them ... um ... so, it’s more about, it’s not to rev ... hang on ... oh [indecipherable]... rather than to reverse, it would be to, place them from highest to lowest.” In summary, 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 think aloud, he did not manifest concrete operational reasoning. Elsewhere (Teague et al., 2013), we have described think aloud sessions where students do manifest more sophisticated modes of reasoning than Donald exhibited.

Our think aloud study has pedagogical implications. From the earliest lessons, the traditional approach to teaching programming places heavy emphasis on code writing and code design. In so doing, the traditional approach effectively skips across the sensorimotor and preoperational stages and assumes that all students can immediately reason about code at the concrete operational stage. Our think aloud with Donald, and think alouds with other students, suggests that the traditional approach skips too quickly across the sensorimotor and preoperational stages.

References


Increasing Student Confidence throughout the Computer Science Curriculum

John A. Trono

Computer Science Department
Saint Michael's College
jtrono@smcvt.edu

Keywords: POP-I.B. barriers to programming, POP-II.B. program comprehension, POP-IV.A concurrency

Abstract
As educators, we all try to do our best to help the students enrolled in our classes learn/understand the concepts, techniques and strategies pertinent to those courses. One effective methodology to aid this pursuit is to devise a semester-long plan of activities that will attempt to increase each student’s level of confidence that they are adequately learning the course’s contents. Experience has shown that students who are confident about their knowledge competency tend to stay more focused and motivated about the subject matter, which tends to improve their retention of this knowledge obtained therein well beyond the course’s duration. The remainder of this report will describe some of the specific, course-related activities that have been used to bolster student confidence in many different computer science courses.

1. Introduction
The basic premise upon which this work is built is something that this author has been incorporating since his first few years as an educator (almost thirty years ago). From those early days as a novice, assistant professor, to the present day evolution of this ‘sage on the stage’, one thing has always been clear to me as I’ve looked at the faces of those attending my course presentations: students who believe that they are understanding the concepts being described are more engaged, active learners while attending course lectures than those whose facial expressions actually remind me of ‘a deer caught in the headlights’.

It is unimaginable that one could meet a teacher, at any level – in any system of education, who can say that they have never seen that look of sheer terror that is concomitant with the realization that what the speaker is trying to convey makes no sense whatsoever to that person who is listening to them. Once that feeling of being lost has been recognized by the listener’s brain, one of two things usually occurs: either a hand is raised, and a question is asked (to rectify the situation), or, that student continues to sit there, only listening passively while taking notes, hopeful that a revelation will occur to them at a later time – perhaps while they are reading passages from the assigned, course textbook.

Unfortunately, many students who realize that they are in a course which appears to them to be beyond their ability to completely comprehend, sadly acquiesce into ‘survival mode’, if it is too late to withdraw from this course. In this mode, the student will work as hard as they can, and they will hope that they will receive at least a passing grade in the course so that they will receive the credits for completing the course, and it will therefore not have been a total waste of their time. How sad is it that some students must rely on this approach at all, especially since becoming a more informed citizen is one important outcome of higher education – not just the semester’s grade point average, or the eventual diploma that may be earned.

Thankfully, many intelligent, motivated students resist this temptation and persevere past such roadblocks. It has become clear to me over the years that all those who do overcome such temporary setbacks typically seem to share one common trait: they all believe in themselves, and are confident in their ability to learn new ideas. For some students, their confidence can be lessened during their time in a course, and if it is, their level of engagement slowly ebbs towards passivity, and their internal push to ask clarifying questions during class decreases until it is non-existent (for that course).
This self-defeating, reinforcing behaviour is usually supported by two faulty student assumptions. First, “… the teacher will think that I am stupid because I have to ask them questions”; students have said that they believe the teacher will also “… assume that I think that they are a terrible teacher since I can’t follow what they are doing in class.” Secondly, “If I ask a stupid question”, it will allow “… the teacher to become aware how low my IQ is, and I am hoping to hide that fact from them.”

Some students have such strong identities and personalities that it would be difficult for one teacher (in one course) to alter their self-esteem at such a late developmental stage. However, many students are not so strongly aware of their own inherent capabilities, and those are the ones who would benefit the most from the techniques and methodologies suggested in this paper.

Subsequent sections will outline strategies (to increase student confidence) that have been integrated into courses on software development, operating systems, computer architecture, and computer networks, and hopefully this report will inspire those who read it to develop more of the same.

2. Introductory Level Programming Courses

Many universities did not have programming courses in the 1960’s, but now that the information revolution has taken hold, such courses are available to many high school students – or even younger individuals, if they wish to learn about how to instruct a computer (to do what they want it to do) either by reading a book on that topic, or by some online, educational experience.

Most college curricula now have incoming students enrol in a basic course that introduces them to problem solving on the computer, implementing their software artifacts in some high level, programming language, e.g. Java, C++ or Python. Some students may test out of this course in some fashion, most popularly in the United States by having successfully scored high enough on an Advanced Placement (AP) test, after having completed an AP course in high school. (Many colleges state-side will give college credit if a student earns a 4 or 5 – out of 5 – on an AP exam.) Most institutions then follow this course with one where data structures and algorithm analysis are the primary topics of interest.

At Saint Michael’s College, our computer science (CS) curriculum has three courses in the introductory software development sequence. Our first programming course (CS111) is essentially open to all students, so we enrol quite a variety of students in it, from many different backgrounds and majors, as CS111 also fulfils the quantitative reasoning, graduation component within the College’s, required Liberal Studies Curriculum. The following course, CS113, focuses more on ‘programming in the large’ since the largest program a student will complete in our CS111 course will most likely be between 50 and 100 lines of Java code (not counting comments). Hence, the two primary goals of CS113 are: to expose our students to some of the software engineering techniques that are used by many professionals when they develop software; and to help our students reinforce (and extend) their own problem solving skills as they are designing (and implementing) programs composed of hundreds of lines of code (and possibly many, distinct Java classes/objects).

CS113 also provides the students (who take it) with an opportunity to increase their confidence in the programming skills that they acquired in CS111 since those skills are now going to be reinforced (and enhanced) by the completion of more challenging and intricate programming projects in CS113 (than those that can reasonably be assigned to students taking CS111). This curricular decision has been an important building block that was instituted back when our department was created (1982), and we believe that it has been a very valuable, pedagogically sound choice that has aided many students – especially those who came to our program with no prior programming experience whatsoever.

One technique that I’ve always found to be quite effective with regards to reinforcing student understanding, when I am teaching our CS111 course, is illustrating how the values of the variables change (over time) while a program is executing. Mayer (1981) cites the results from an earlier study of his where students, that were exposed to his “concrete model”, performed better on questions pertaining to basic program understanding that those in the control group. More recently, Msele (2010) has verified that the use of his RAM diagrams have likewise been a valuable methodology for enhancing comprehension (when teaching programming).
Both of these approaches are slightly different manifestations of having students carefully trace a program in a step by step manner. I also demonstrate this technique during my lectures, though I simply list each declared variable in a horizontal fashion, with a ‘?’ underneath each one, to show that its value is currently unassigned. Then, each time a variable is updated during an assignment statement, the value in the corresponding column is modified to indicate that current value (which has been placed into that memory location, for that specific variable).

Because repetition is a very powerful tool for those attempting to master the skills in CS111, my students will trace (and write) programs on weekly quizzes as well as during the three, in class exams given each semester; there are also weekly programming assignments that will help them to internalize the art and science of computer programming. Both skills – tracing and creating software – are fundamentally what CS111 is all about. However, learning how to correctly trace a program is quite an important skill since it allows the students to determine why their program’s output is incorrect, which can help them when creating a program on a quiz or exam, since it provides them a technique to discover any logic errors – and rectify them before the result is assessed.

Another useful tool in these introductory CS courses is the concept of an extended, or multi-phase, assignment. Huggins et al (2003) describe several places where having a project contain several phases, which may be given as consecutive assignments, or can be spread out throughout the semester, can be beneficial. Student confidence seems to build during the time these related assignments are being worked on (whenever I’ve utilized this approach).

More recently, Chen and Hall (2013) have also used the last month of a first programming course to expose their students to a lengthy, cumulative programming project, with several deadlines and milestones spread throughout that time period. A student quote indicates how positive this assignment was for them: “This project was a great learning experience … I now realize that I’m capable of making fairly complex programs … and because of that, I feel confident and excited to program more.” The course instructors also seem to concur with this belief: “We observed an increase in confidence amongst the students as they progressed through the project.”

Textbooks have a great impact on what students learn during their courses, and many books at the introductory level attempt to build their reader’s confidence regarding their internalization of those particular concepts. One popular strategy to accomplish this is to sprinkle review/self-help questions throughout each chapter rather than place them at the end of each chapter. Using the former approach, students can gain immediate feedback and assess their level of understanding while they try to comprehend the material. This self-assessment strategy is facilitated by incorporating the answers to these questions in an easily accessible place so students can validate their newly acquired knowledge, or, they can return to previous sections in the book to uncover what they missed during their initial pass over those ideas. (I’ve noticed recently that some upper level CS textbooks are beginning to do likewise as well.)

This section has included several strategies that may help to augment each student’s learning of the desired, basic programming skills. Because these skills must be retained for subsequent CS courses, it is important that they are not forgotten. The more hands on experiences that each student successfully completes, the more likely it is that these skills will become like ‘second nature’ to those students who complete the course.

To make the more abstract concepts of advanced, upper level CS courses as crystal clear as possible, to all the students enrolled in those classes, it is also important to provide many hands on exercises and projects to bolster their understanding as well. The following three sections will enumerate what approaches seem to have benefited my students the most over the years (in this regard).

3. Understanding the Mechanisms inside an Operating System

Mayer (1981) stated “… understanding is defined as the ability to use learned information in problem-solving tasks that are different from what was explicitly taught … ability to transfer learning to new situations.” This is a point that I stress at the first meeting of a required junior/senior level course that focusses on operating systems (O/S) concepts. As an example, I show them how to add
two three digit numbers, illustrating the process of propagating the carries, e.g. \(468 + 795\). I then proceed to say that I now expect that they could add two seven digit numbers in the same manner, if they truly understand the process, even though I have not ever explicitly demonstrated a specific example of that size.

As advanced, upper level undergraduates, these students need to be prepared to become lifelong learners, and to be more independent in advanced courses like this one since they will be expected to continue to develop professionally, on their own, after graduation. That is why I tell them that they can’t rely on memorization, for the same reason that that strategy doesn’t work when trying to learn how to program; one must understand what each statements does, and how to select the right ones – in the correct order – to solve the problem at hand. Remembering the exact statements in one program may be useless when solving an entirely different problem.

To emphasize this point, there must be less hand holding in courses like this one, than in CS111: typically, upper level students are provided one opportunity to demonstrate proficiency on a topic, unlike in CS111, where quiz questions are a preparatory step for similar questions on upcoming exams, and these exams are weighted more heavily, since students should be rewarded if they’ve finally learned the material even if initially they were ‘not quite getting it’. (This approach seems reasonable because the student mix in CS111 is also more heterogeneous than in an advanced course like one investigating O/S concepts.)

Homework assignments, quizzes and programming projects are all certainly quite helpful tools as students attempt to understand the O/S topics presented, but besides exercises on short term scheduling strategies, local page replacement algorithms and deadlock avoidance techniques, for example, several other useful instructional aids will now be described.

Concurrency, the concept of more than one program being necessary (to efficiently solve a problem), has always been an important topic that ties together many of the underlying O/S concepts (after those concepts have been described earlier in the course). To help students gain confidence in this area, one that is quite different from the straightforward, sequential programming that they’ve been exposed to up to this point in time, I provide them with many homework problems to solve, all of which require them to construct process pairs (via paper design) that utilize semaphore operations to control their independent (and possibly parallel) execution.

These homework problems are assigned in such a way that one is due at the beginning of every class for roughly six lectures. Students are also informed that the first few of these assignments will be graded more leniently than the ones that follow, since concurrent programming is probably new to them. Problems that are handed out in one class period aren’t due the next period (when they can ask for clarification on the problem), but their solution must be turned during the class after that one.

Students are told that for each such homework, one submission will be selected randomly, and it will be copied onto the board for inspection. (Students will only have at most one of their submissions publically displayed in this manner.) My experience is that students certainly learn from how a correct solution solves the posed, concurrent programming problem, but they learn more from recognizing what does not work in someone else’s proposed solution. Using this approach, many students can also learn what to avoid by seeing such demonstrative examples. Students will also be asked to solve a similar problem on a quiz (and on the final exam), and one of these exercises will be chosen for direct implementation, using the semaphore primitives that are available on our Linux-based workstations (in our upper level, computer lab).

One final piece of the ‘concurrency learning experience’ is to have the students be exposed to other related topics. One strategy that I’ve found to be beneficial is to discuss how general counting semaphores can be implemented using the binary semaphore primitives if they are available on the underlying hardware, and the problems in one such implementation have been documented. I have distributed the relevant articles that have described the problem, and the debate that followed, and then require the students to determine which of the arguing authors view is most correct, and why, and to place their views on this debate in a paper of their own creation. (A brief introduction to the problem, and the articles that compose this debate, can be found in Trono and Taylor (2000).)
4. Computer Architecture: Enhancing Understanding through Performance Analysis

Our CS majors typically enrol in our required computer architecture course in the semester following the O/S course, which is also roughly one year after our students complete our machine organization course. Back in the 1980’s, there were truly no outstanding computer architecture textbooks, but when the first edition of the popular Hennessy and Patterson book (Computer Architecture: A Quantitative Approach) was released in 1990, I adopted it, and have used it ever since. Even though many topics that I consider essential to what I emphasize in this course have been moved into the book’s appendices, I believe it is still the top text for exposing CS students to the level of detail necessary to understand how a processor executes the program’s instructions, and what techniques are used to reduce the time it takes for the programs to complete.

Though this book includes many pertinent exercises at the end of each chapter, I’ve created my own extended assignment that illustrates how much time is saved by the architectural changes that have been incorporated into microprocessors over the years. More specifics about this collection can be found in Trono (1999), but the basic outline is as follows. A typical program in the SPEC (Standard Performance Evaluation Corporation) benchmark suite was chosen, written as a C program, and then translated into a reasonably well optimized, MIPS assembly language (where MIPS is the instruction set architecture described in Appendix C of the 5th edition of Hennessey and Patterson’s book).

The first assignment is to familiarize the students with the C program, one that they will continue to analyse throughout the semester, and to simply calculate the total number of machine cycles it will take to execute this iterative code stub (64K iterations, i.e. 65,536 executions of the code within that loop) with two underlying, instruction execution models. This basic architecture is then improved upon in the next assignment, using the same MIPS code, by first adding a two stage pipeline, and then adding a finite, instruction look-ahead buffer, and the performance improvements can then be quantified using the relevant formulas from the book’s opening chapter.

Next, the inclusion of a data cache, and then an additional instruction cache, reduce the memory bottleneck’s impact on the program’s performance. Finally, the C code is once again translated to MIPS, this time utilizing the available vector instructions present, and only 1024 iterations are needed since each iteration now processes data vectors of 64 elements at a time. The CPI (average number of cycles per instruction) is measured for the architecture in each of these assignment extensions, and the resultant speedup values can be computed, for comparison purposes, after each hardware improvement is introduced. Students do become more confident as this assignment unfolds during the semester, and their performance on these homework problems improve over time even though each subsequent problem is more complex than its predecessor.

Several other possible hardware improvements could be used for subsequent assignments after caches are added, like including a scoreboard, or reservation stations, to facilitate an out of order execution strategy that could extract more instruction level parallelism from this program. However, since the students are also performing research on a topic from a paper chosen from a recent International Symposium on Computer Architecture (ISCA) at that time, I have chosen not to overload them with more homework assignments in the last quarter of the semester when they are feverishly working on their paper, which they could only begin in earnest after the midterm exam. (Their topic was selected early in the semester, so they could collect the relevant references during the first half of this course. This assignment comprises 20% of their overall grade, and must be submitted in written form.)

5. Computer Networks: Packet Flow Activities

The primary objectives I have, when teaching this particular course, is for the students to come away with a deeper understanding concerning how packets travel over the Internet, especially during the transmission of large music or picture files, as well as how basic mechanisms within the protocol stack actually work. Following one idea that permeates this paper – that repetition is one major key to learning, and retaining, how to do something, especially because reinforcing the steps needed to accomplish a task helps to cement those steps in one’s mind – the principle that ‘if you don’t use it, you lose it’ is one that seems to be worth keeping at the forefront of one’s plan for the semester, when
establishing a timeline for what should occur during the limited amount of time allotted to enlighten the enrolled students about the topics to be covered.

Error detection (and correction) is an important concept to convey, and cyclic redundancy check (CRC) codes, as implemented in the data link layer, are probably the primary methodology to accomplish this task in many popular protocols. CRC codes also can be efficiently implemented in hardware, using shift registers, and exclusive-or operations, so this topic combines and revisits ideas presented in the machine organization course that students have recently completed. CRC codes also lend themselves to repetition in homework, quiz and exam questions, for reinforcement purposes, and they allow for the investigation of one way to create an error-free conduit for packets/frames to travel.

How packets are routed (between the original and final destination nodes) is another area where each student’s confidence (of the knowledge they are acquiring) can be increased through opportunities that reinforce what they have learned via assessment tools like homework, quiz and exam questions. Several different, distance vector problems are assigned as homework, and once these algorithms have been mastered, the next idea to investigate is how congestion control in the Internet is accomplished. (Congestion occurs if too many packets are attempting to be sent, and the available bandwidth is not high enough to prevent packets from being dropped, due to insufficient space to buffer the excess packets that are queued up for transmission to the next node – in their journey to the final destination specified.)

Many examples of how TCP Tahoe works are presented in class, and as homework/quiz problems, and then other implementations like TCP Reno, and/or other more recent modifications to this basic strategy can be covered, like Selective Acknowledgements (SACK), allowing students to get a better sense of what is going on behind the scenes.

6. Conclusion

While no strategy will work perfectly with every student, my experience has been that the in class strategies, and individual assignments, described here have been reasonably successful in helping students recognize that they have acquired the basic understanding of the related, specific topics (in their respective computer science courses). Ensuring that each student’s confidence in their ability (at a high level) seems to help them remain engaged learners in the classroom. This in turn, I believe, helps them to stay focussed and alert during class meetings and eventually gives them the best chance to successfully complete the course – and to maximize their understanding of the concepts and techniques presented.

7. References


The cognitive difficulties of Description Logics

Paul Warren*  
paul.warren@cantab.net

Trevor Collins*  
trevor.collins@open.ac.uk

Paul Mulholland*  
paul.mulholland@open.ac.uk

*Knowledge Media Institute, The Open University, UK

Keywords: POP-I.A Description Logics, POP-II.B program comprehension, POP-III.B OWL, POP-V.A mental models

Abstract

This paper describes work-in-progress investigating the difficulties experienced in using Description Logic constructs. In particular it views these difficulties from the standpoints of cognitive dimensions, mental models and relational complexity. It outlines how the latter two theories might be applied to estimate the cognitive complexity of individual Description Logic constructs and patterns of constructs. The paper also describes some initial feedback obtained from users of ontologies and Description Logics.

1. Introduction

The use of ontologies based on Description Logics, in particular the various profiles of the Web Ontology Language (OWL), is now prevalent in a variety of domains, including biomedicine, e.g., SNOMED\(^1\); engineering, e.g., the Semantic Sensor Network Ontology (Compton et al., 2012); and business, e.g., Good Relations\(^2\). OWL is also one of the languages standardized by the World Wide Web Consortium for use in the Semantic Web. Description Logics have been studied extensively from the viewpoints of ensuring decidability and also of their general computational tractability, e.g., see Donini et al. (1997). However, there has been much less work to investigate their usability by the producers and consumers of ontologies. In particular, little has been done to understand the cognitive difficulties encountered in using Description Logics. This paper will describe work-in-progress investigating the difficulties experienced with Description Logic constructs and seeking to explain those difficulties in terms of cognitive science.

The next section describes some related work in this area. Section 3 then briefly describes the results of a survey of ontology users. Sections 4, 5 and 6 in turn apply the frameworks of cognitive dimensions, mental models and relational complexity. Finally, section 7 draws some conclusions and outlines future work.

2. Related Work

There has been some related work by those who are interested in helping ontology builders debug their ontologies. Typically in this situation, when automatic reasoning is applied, errors in ontology design lead to unexpected and unwanted inferences. To aid debugging, users are presented with a set of axioms selected from the ontology to justify the inference.

Horridge et al. (2011) undertook experiments with computer science students to study their understanding of justifications expressed using logical symbolism, as they might be presented to aid debugging. They compared their results with their own model of complexity, which was not grounded in any cognitive theory. They found that in most cases there was a significant difference in student errors between justifications ranked easy and hard by the model.

Nguyen et al. (2012) were similarly motivated to understand how to best explain inferences to ontology builders. They did not propose a model of complexity but undertook experiments to establish, for a set of 51 deduction rules, the probability of each rule being correctly understood.

---

1 http://www.ihtsdo.org/snomed-ct/
2 http://www.heppnetz.de/projects/goodrelations/
These deduction rules were regarded as single inference steps, and the intention was that the probabilities could be used to decide between different proof trees. A subsequent experiment confirmed that the model worked relatively well for predicting the probability of correctly understanding proofs consisting of two inference steps (Nguyen, Power, Piwek, & Williams, 2013). In this work, unlike that of Horridge et al., the deduction rules were presented in English and the experimental participants were drawn from a general sample of the population.

3. Ontology Users’ Survey

As an initial study, a survey has been undertaken of ontology users (Warren, 2013). The survey was aimed at ontology users in general but included questions specifically relating to Description Logics. Over a hundred responses were received. The survey responses include information about the size of ontologies; use of visualization tools; use of ontology patterns and pattern tools; Description Logic features and OWL profiles in use; and the choice of ontology tools. The response to a question about ontology editors indicated that the most popular editors were based on the use of Description Logics. This indicates a shift from an earlier frame paradigm. In the frame paradigm the ontology developer defines classes and relations between instances of classes; using Description Logics relations can also be used to define classes. Some of the comments received suggest that whilst there are a number of people skilled in the use of sophisticated Description Logic constructs there are also others who find this difficult; e.g., one respondent commented “The rigor of the languages exceeds the rigor of the typical user by a wide margin”.

4. Cognitive Dimensions

The cognitive dimensions framework was developed to assess the cognitive difficulties inherent in a wide range of cognitive artifacts, e.g., see Green and Blackwell (1998) for a tutorial and Green and Petre (1996) for an example of an application of the framework. Figure 1 illustrates, in broad terms, how this framework might describe the use of Description Logics to create ontologies. Some of these comments relate to ontologies generally, e.g., the limited features for abstraction. A number of the comments relate to the available tools. In general the state-of-the-art in ontology development tools lags that of software development tools; the need for improved tool functionality was a comment made by respondents to the ontology users’ survey mentioned above.

There are four dimensions which are specifically relevant to the use of Description Logics, as distinguished from ontologies generally or current tools:

- **Hard mental operations.** Description Logics are based on subsets of first order logic. This raises two problems. Firstly, humans are not adept at handling logic. Secondly, the usage of language does not always correspond to everyday usage. The most commonly used linguistic representation is Manchester OWL Syntax (Horridge et al., 2006). Whilst apparently English-like, this syntax can lead to confusion.

- **Hidden dependencies.** Even relatively small sets of axioms may contain hidden dependencies. Typically these are not apparent until an automatic reasoner is executed, when unexpected and possibly unwanted inferences become apparent, as noted in section 2.

- **Diffuseness, role expressiveness and error-proneness.** The language used is very terse, being based on logic. This avoids ambiguity but, combined with the unusual English usage, creates difficulties of comprehension. For example, the expression ‘X SubClassOf R some Y’, where X and Y are classes, R is a relation, and ‘some’ is a keyword, is equivalent to the more normal English usage ‘every member of X has the relation R to some member of Y’. The latter, more verbose usage, may be regarded by some people as more intelligible. The use of the ‘only’ keyword to represent the universal quantifier, combined with the use of relations in Description Logic to define classes, demonstrates the tendency to error-proneness. The difficulty here is that in logic the universal quantifier can be trivially satisfied, i.e. ‘∀x ∈ X: P(x)’ is true for all predicates P when X is the empty set; this does not correspond to the normal English usage. This difficulty can be illustrated with an example taken from Nguyen et al. (2013). The statement...
“a good movie is anything that has as ratings only four stars”, if interpreted as a Description Logic statement, defines the class ‘good movie’. This class not only contains movies rated with four stars, but anything which does not have a rating, including things which are not movies.

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Limited abstraction mechanisms. (Applicable to ontologies generally).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closeness of mapping</td>
<td>Very close mapping of logical relations; difficulties occur when additional features, e.g., arithmetic, are required.</td>
</tr>
<tr>
<td>Consistency</td>
<td>Some opportunity for alternative but equivalent formulations, e.g., a relation can be declared as symmetric or being its own inverse.</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>Very concise. (This is generally true of ontology languages).</td>
</tr>
<tr>
<td>Error-proneness</td>
<td>Some conventions different from everyday use of language; most notably the ‘trivial satisfaction’ of universal quantifier, represented by ‘only’ in Manchester OWL Syntax.</td>
</tr>
<tr>
<td>Hard mental operations</td>
<td>Places high demands on human reasoning.</td>
</tr>
<tr>
<td>Hidden dependencies</td>
<td>Automatic reasoner can produce ‘unexpected’ inferences, as a result of logical relationships.</td>
</tr>
<tr>
<td>Juxtaposability</td>
<td>This is a feature of the tool; not generally present in common tools.</td>
</tr>
<tr>
<td>Premature commitment</td>
<td>No ordering of operations specified.</td>
</tr>
<tr>
<td>Progressive evaluation</td>
<td>Automatic reasoner can be run at any time.</td>
</tr>
<tr>
<td>Role-expressiveness</td>
<td>Concise logical language makes comprehension of component roles difficult.</td>
</tr>
<tr>
<td>Secondary notation</td>
<td>Graphics and colour can be used; visualization techniques are common.</td>
</tr>
<tr>
<td>Viscosity</td>
<td>Ontologies can be designed modularly to limit effects of changes</td>
</tr>
<tr>
<td>Visibility</td>
<td>Hard to view large ontologies</td>
</tr>
</tbody>
</table>

![Figure 1 – application of cognitive dimensions framework to ontologies and Description Logics](image)

5. Mental Models

The mental model framework was developed to explain the difficulties people have with logical reasoning. For an overview, see Johnson-Laird (2005); for an application to disjunctive reasoning, see Johnson-Laird et al. (2012). The essence of the theory is that we reason not by applying logical transformations as an automatic reasoner would, but by enumerating possibilities. The theory distinguishes between fully explicit models and implicit mental models; the latter are frequently referred to simply as mental models. In fully explicit models, all possibilities are enumerated. In mental models, which are more typical of human reasoning, the focus is on statements which are true.

The number of possibilities evoked by the mental model theory, at each step of a chain of inferences, may provide a measure of the cognitive complexity of each step. At the same time, those possibilities which are lost in the transition from the fully explicit to mental model may account for errors in
reasoning. Figure 2, which is taken from Johnson-Laird (2005), illustrates the two types of models for a conditional statement. The tendency is to focus on the case in which A and B are true and neglect, for example, that B can be true when A is false.

<table>
<thead>
<tr>
<th></th>
<th>Fully explicit model</th>
<th>Mental model</th>
</tr>
</thead>
<tbody>
<tr>
<td>If A then B</td>
<td>A, B</td>
<td>A, B</td>
</tr>
<tr>
<td>~A</td>
<td>B</td>
<td>...</td>
</tr>
<tr>
<td>~A</td>
<td>~B</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 – fully explicit model and mental models; from Johnson-Laird (2005)

Figure 3 illustrates how the mental model approach might be applied to a simple example from Description Logics. The left-hand side represents the fully explicit model for each axiom; the right-hand side represents the mental models and shows how they can be combined. In the figure X, Y and Z are classes and x, y, z are representative members of those classes. R represents a relation. The first axiom is ‘X SubClassOf R some Y’. The significance of this statement was explained in section 4. This gives rise to two possibilities; that the representative element x of X is in relation R to a representative element y of Y; and that there is another representative element of Y for which there is no X in relation R. This second possibility may be omitted by the human reasoner because it is not the focus of attention. The second axiom is similar. The third axiom is the transitivity of R. This axiom is applicable to all individuals in the domain of discourse, not just the elements of X, Y and Z. However, here the focus is on these three classes. If we consider two relations a R b and c R d, where a, b, c and d can be individuals in any of X, Y, Z, then we have 81 (3^4) possibilities in the fully explicit model. To simplify, in the figure we ignore ordering (so that a R b c R d and c R d a R b are equivalent), omit the repetition of identical relations (a R b a R b) and omit the repetition of an individual in any particular relation (a R a). This gives a total of ten possibilities^3, as illustrated. The right-hand side of the figure shows the simplification when we focus on the apparently most significant possibilities, i.e. those for which the transitive relation leads to a conclusion.

X SubClassOf R some Y; Y SubClassOf R some Z; R Transitive => X SubClassOf R some Z

Fully explicit models

first axiom:

x R y

second axiom:

y R z

third axiom (transitivity):

x R y y R x x R x
x R y x R z
x R y y R z x R z
x R y z R x
x R y z R y
y R x x R y y R y
y R x x R z y R z
y R x y R z
y R x z R x
y R x z R y

Mental models

first axiom:

x R y

and second axiom:

y R z

combine to:

x R y y R z - (A)

third axiom (transitivity):

x R y y R x x R x
x R y y R z x R z
y R x x R y y R y
y R x x R z y R z

combining (A) and third axiom gives:

x R z

Figure 3 – fully explicit models and mental models for a Description Logic inference

^3 Without loss of generality, we assume that the first relation relates x and y (x R y and y R x). Then there are five possibilities for the second relation: the unused relation between x and y plus x R z, y R z, z R x and z R y. This gives rise to 2 x 5 = 10 possibilities.
6. Relational Complexity

Halford et al. (2010) argue that each step in a reasoning process is characterised by an n-ary relation. Formally an n-ary relation is a set of ordered n-tuples. Thus the statement ‘Bob is taller than Tom who is taller than Peter’ is an instance of a ternary relation which contains the 3-tuple (Bob, Tom, Peter), amongst others. Proportion, e.g. ‘3/5 > 4/7’, entails a quaternary relation, see Halford et al. (1998). It is argued that human processing capacity is limited to one quaternary relation in parallel. When confronted with relations for which n>4, we have recourse to strategies such as chunking variables together and breaking complex relations down to be processed in sequential stages, rather than concurrently. In their ‘method for analysis of relational complexity’, Halford et al. (2007) propose that the processing complexity of a multi-step task is determined by the relational complexity of the most complex step.

Figure 4 shows how this might be applied to the example of the previous section. The statement ‘X SubClassOf R some Y’ is an instance of a ternary relation which can be represented by S(X, R, Y). The second axiom can similarly be represented by S(Y, R, Z), and the conclusion by S(X, R, Z). The third axiom represents a ternary relation because there are three slots in a transitive relation. It is also possible that it will be perceived by many people as a unary relation, T(R); this is an example of chunking to reduce dimensionality of a relation.

Part of the difficulty with expressions of the form ‘X SubClassOf R some Y’ is that ‘R some Y’ is to be understood as the anonymous class containing all individuals that are in the relation R with an individual in Y. A more natural English usage is ‘every x R y’. Here the focus is on the individuals x and y rather than the classes they represent. Perhaps more importantly, by chunking the ‘every’ and the ‘R’ this statement can be perceived as a binary relation either between the representative individuals x and y or the classes X and Y, written P(X, Y). The latter is illustrated in figure 5.

\[
\begin{align*}
X \text{ SubClassOf } R \text{ some } Y & \quad \text{ternary relation} = S(X, R, Y) \\
Y \text{ SubClassOf } R \text{ some } Z & \quad \text{ternary relation} = S(Y, R, Z) \\
R \text{ Transitive} & \quad \text{ternary relation involving three slots or unary relation} = T(R) \\
\Rightarrow X \text{ SubClassOf } R \text{ some } Z & \quad \text{ternary relation} = S(X, R, Z)
\end{align*}
\]

Figure 4 – Description Logic inference regarded as a set of n-ary relations

\[
\begin{align*}
\text{Every } x \ R \ y & \quad \text{binary relation} = P(X, Y) \\
\text{Every } y \ R \ z & \quad \text{binary relation} = P(Y, Z) \\
R \text{ Transitive} & \quad \text{unary relation} = T(R) \\
\Rightarrow \text{Every } x \ R \ z & \quad \text{binary relation} = P(X, Z)
\end{align*}
\]

\[P(A,B) \text{ is equivalent to } \text{‘Every } A \ R \ B\text{’; effectively chunking together relations ‘every’ and ‘R’.} \]

Figure 5 – the effect of language on relational complexity

A graphical representation can be used to compare the complexity of the approaches of figures 4 and 5, in line with a long history of representing logic graphically, e.g. see Stenning and Oberlander (1995). When represented graphically, the introduction of the anonymous classes in the approach of figure 4 can be seen to lead to extra complexity.

7. Conclusions and Future Work

This paper has described how the problem of understanding the cognitive complexity of Description Logics can be viewed from three different frameworks. Cognitive dimensions provides a general view of where the difficulties lie in interpreting Description Logics, and set this in the context of the problems experienced with ontologies generally and of the current state-of-the-art of available tools. Mental models provide a means of interpreting Description Logic statements as sets of possibilities.
Relational complexity focuses on the complexity inherent in the relations employed at each step of the inference chain. In both cases the predicted limitations on reasoning have their roots in what can be held concurrently in working memory. The mental model and relational complexity theories now need to be applied to predict the difficulty of a range of Description Logic inferences; these predictions can then be investigated by experiment. Graphical models can also be applied to help understand logical complexity. It is hoped that this will lead to a model of the comprehensibility of Description Logic statements which will have a number of benefits: a clearer understanding of where the problems lie, and how they can be avoided by choice of logical constructs; better representation of these logical statements in a more natural language; and improved tools, methodologies and training.

8. References


Towards Defining a Cognitive Linguistics of Programming and Using Eye Tracking to Verify Its Claims

Sebastian Lohmeier
Technische Universität Berlin
sl@monochromata.de

Keywords: POP-II.B program comprehension, POP-V.A cognitive linguistics, POP-V.B eye tracking

1 A Cognitive Linguistics of Programming

There are cognitive models that explain programming (Parnin, 2010) and psychological experiments based on models from cognitive psycholinguistics (Burkhardt, Détienne, & Wiedenbeck, 1997). In computer science, attempts have been made to enhance programming languages based on knowledge of natural languages (Knöll, Gasiunas, & Mezini, 2011; Knöll & Mezini, 2006; Lopes, Dourish, Lorenz, & Lieberherr, 2003). Combining these three relations, I propose a cognitive linguistics of programming that is based on three-level semantics, a theory of text comprehension from cognitive linguistics (Schwarz, 1992; Schwarz-Friesel, 2007). The cognitive linguistics of programming enables psychological experiments and is used to enhance programming languages.

Three-level semantics was chosen as the basis of the cognitive linguistics of programming, because it specifies structures and processes that can be implemented computationally and are comparable to those found in object-oriented programming languages. Furthermore, three-level semantics is suitable for generating testable psychological hypotheses and incorporates the reader. That is why I describe a cognitive linguistics of programming, not of programming languages: three-level semantics assumes an active role of the reader – she constructs meaning while reading. Using such a reader-centric model of language, it becomes possible to explain problems related to programming language understanding during the act of programming.

To make a start, the cognitive linguistics of programming is realized by adding indirect anaphors known from linguistics to the Java programming language (Lohmeier, 2011).

2 Indirect Anaphors in Linguistics

How are indirect anaphors described in linguistics? Example (1) contains the indirect anaphor the Expression whose definite article the signals that the Expression is known, either due to world knowledge or prior mention.

(1) An if-then statement is executed by first evaluating the Expression. If the result is of type Boolean, it is subject to unboxing conversion (Gosling et al., 2005, 372)

According to Schwarz-Friesel (2007), the indirect anaphor the Expression is understood in relation to its antecedent An if-then-expression and based on the knowledge that if-then statements contain an expression. Using conceptual schemata, this fact is expressed as a part-whole relation between EXPRESSION and IF-THEN-STATEMENT (upper case denotes concepts). Reading the sentence can then be described as follows: While reading the antecedent An if-then-statement, the default EXPRESSION part of the IF-THEN-STATEMENT concept is activated in the mind and is still active when the Expression is read. The definite article the signals that the expression refers to a known EXPRESSION and since the default EXPRESSION part of the IF-THEN-STATEMENT is the only active referent of a matching concept, the expression refers to it. During reading, the reader retrieves from memory the part-whole-relation that is
underspecified in the text, but available from long-term memory. Likewise, she reconstructs that the result is the result of the Expression. By resolving the underspecification of the indirect anaphors the Expression and the result, the reader understands a text that would be longer otherwise:

(2) An if-then statement is executed by first evaluating its Expression. If the result of the Expression is of type Boolean, it is subject to unboxing conversion

If the underspecified relations are well known to the reader, underspecification shortens texts and improves learning of new relations explicated in the text (McNamara, Kintsch, Butler-Songer, & Kintsch, 1996).

3 Indirect Anaphors in Programming

Knowledge of part-whole and other semantic relations is encoded in object-oriented programs. It is thus desirable to shorten source code by introducing indirect anaphors to programming languages in order to improve programmers’ learning from source code. This requires compilers that are able to process indirect anaphors similar to programmers. Listing 1 shows an example of an indirect anaphor in programming.

Listing 1. Indirect anaphor .Result in a modified org.junit.runner.JUnitCore (JUnit 4.8.2)

```java
public static void runMainAndExit(JUnitSystem system, String... args) {
    new JUnitCore().runMain(system, args);
    system.exit(.Result.wasSuccessful() ? 0 : 1);
}
```

The indirect anaphor .Result in line 54 refers to the Result returned by the invocation of runMain, eliminating a local variable from the original source code shown in Listing 2.

Listing 2. Original snippet from org.junit.runner.JUnitCore (JUnit 4.8.2)

```java
public static void runMainAndExit(JUnitSystem system, String... args) {
    Result result = new JUnitCore().runMain(system, args);
    system.exit(result.wasSuccessful() ? 0 : 1);
}
```

4 Claims

Experienced programmers should not have difficulty understanding both listings. While it is not clear whether they would take less time to read the code, if the findings for underspecification in natural language hold for indirect anaphors in programming languages, programmers can be expected to better recall new information from source code underspecified with indirect anaphors. Like in the case of underspecification in natural language, indirect anaphors will of course impede understanding for those programmers who do not possess the underspecified knowledge. Identifying what knowledge is available to programmers so as to be able to present code with or without indirect anaphors would require use of a cognitive architecture (Hansen, Lumsdaine, & Goldstone, 2012; Lohmeier & Russwinkel, 2013) and is beyond the scope of this work.

5 Eye Tracking

To test whether source code with indirect anaphors is slower or faster to comprehend than traditional source code, eye tracking could be applied relatively naturally by integrating eye
tracking into an existing IDE (see http://monochromata.de/eyeTracking). This would allow a comparison of eye movements and reading times for indirect anaphors in natural language texts (Garrod & Terras, 2000) and source code. Presenting programmers source code with varying numbers of indirect anaphors would permit comprehension studies in the fashion performed for natural language texts (McNamara et al., 1996). For both reading times and comprehension of indirect anaphors in source code, the effect of a reader’s possession or lack of knowledge required to overcome underspecification will be of interest.

References


Representational Formality in Computational Thinking

Alistair G. Stead  
Computer Laboratory  
Cambridge University  
ags46@cam.ac.uk  

Alan F. Blackwell  
Computer Laboratory  
Cambridge University  
Alan.Blackwell@cl.cam.ac.uk

Keywords: POP-II.A. novices, POP-VLE. computer science education research

Abstract
This paper describes our thoughts on a new addition to the set of Computational Thinking abilities: Representation Expertise.

1. Background
The notion of Computational Thinking (CT), defined first by Jeanette Wing (2006) led to spirited discussions within several research communities on the origins of problem solving, what exactly constitutes thinking computationally, and whether the term has any real world value. The various lists of CT “abilities” (The National Academy of Sciences, 2011) (ISTE, 2013) (Google, 2013) cover many activities related to thinking computationally, such as data representation, problem decomposition, visualisation, parallelisation and others.

Despite being a core component of computer science, Wing (2006) (2008) intentionally avoided describing “programming” as an ability of CT. Although this is consistent with the original aims of CT: to teach skills and strategies that are transferrable to other disciplines, the exclusion of programming conveniently avoids inheritance of any stigma associated with the term.

2. Representational Expertise as a CT ability
Despite the exclusion of programming as an ability in the original definition of CT (Wing, 2006), Resnick (2011) has argued that to truly master CT, one must be able to express themselves and their ideas in computational terms and therefore have “fluency” with computational media.

We agree with this view of CT. However, we think there is a missing CT ability that has yet to be addressed: Representation Expertise. If one takes the view, as we do, that all computer users are programmers to varying degrees, where “programming” is the interaction with abstractions mediated by some representational notation (Blackwell, 2002), then to express computation fluently, one must master the use of (a range of) notations.

As with representations in other disciplines, such as mathematics and logic, several representations can specify the same result. It is therefore important that users are able to recognise, interpret, and identify correspondences between notations to allow them to achieve competence with representational systems. In mathematics, representational knowledge was found to correlate highly with success in mathematics education (Gagatsis & Shiakalli, 2004). Representational knowledge is also thought to mediate complex problem solving, transfer of knowledge to novel situations, and understanding of high-level concepts (Heritage & Niemi, 2011). The ability to translate between different representations is also thought to contribute towards success in problem solving in mathematics (Gagatsis & Shiakalli, 2004).

3. Current Teaching Practices
The UK National Curriculum is changing its emphasis from application-oriented “ICT”, towards more technical Computer Science based curriculums. Our interviews with teachers in the past eight months...
suggest that in response to this, Scratch is being introduced at the end of Key Stage 2. Other imperative programming languages such as Alice and Python are introduced later, becoming more similar to professional software development tools as the students move through Key Stage 3 and 4 to ultimately meet qualification requirements for GCSE and A-Level. We are concerned that the focus on purely imperative languages may conflict with the need to establish broader representational expertise; students’ ability to recognise and understand new representations is limited, if such a narrow range of visual formalisms (Nardi, 1993) is used, simply offering varying degrees of block syntax. To increase competence with computational representations, students could experience multiple approaches to programming, including visual formalisms that support imperative, declarative and programming by demonstration paradigms.

The degree of formality in computational representations may also vary (Blackwell, Church, Plimmer, & Gray, 2008). For example, a student could express a computational concept as a sketch with pen and paper, which although an informal representation, might be easily decoded by other human readers. We suspect that a major barrier to programming, and to representation expertise, is to understand that computers “think” in a formal manner, and that the specifically formal attributes of mathematical or programming representations may be hard to acquire for this reason. A key skill that has not yet been emphasised as a CT ability is the need to translate one or more informal representations (on paper, or even expressed verbally) into a formal representation the computer can process.

4. Proposed Research

We propose to take standard web technologies as a starting point to investigate these questions. Standard website construction typically involves a mix of declarative and imperative representations, and website design practice often moves from informal sketches to executable models or code. We are creating a tool that can be used in classrooms to provide a far more structured experience, scaffolding the development of formal representation skills.

5. Bibliography


Towards a Mechanism for Prompting Undergraduate Students to Consider Motivational Theories when Designing Games for Health

Monica M. McGill  
*Department of Interactive Media  
Bradley University  
Peoria, Illinois USA  
mmcgill@bradley.edu*

Keywords: POP-V.A. theories of design, POP-I.B. teaching design

1. Introduction

Games are increasingly being used for more than entertainment and educational purposes, and the spread of games into advertising, marketing, and healthcare continues to grow (Gartner 2011). This latter area is often referred to as serious games, or games that are designed with specific outcomes in mind (Engelfeldt-Nielson, Smith, Tosca 2008). In the area of games for health, these outcomes are often patient-centred.

Due to the recent changes to healthcare and the provisions for which physicians in the United States (US) will be paid in the future, there is a particular emphasis on improving patient outcomes (Report on the National Commission on Physician Payment Reform 2013). The new healthcare initiative is highly focused on patient-outcomes, and physicians will be reimbursed at a higher percentage for achieving better outcomes. Towards this end, the medical community is exploring new and innovative ways to improve patient outcomes, including the potential of games for improving and managing health (Institute for the Future 2012).

2. Problem statement and scope of work

Over the last decade, post-secondary institutions in the United Kingdom and the United States have implemented and started to refine game degree curriculum and instruction (Entertainment Software Association 2012, McGill 2011). These programs and their corresponding courses and modules vary considerably in scope and focus and include areas of focus such as Game Design, Game Technology, Game Programming, and Game Art. Given the various types of games that can be created as well as their multi-disciplinary nature, teaching games purely for entertainment is challenging. Designing for serious games adds a layer of complexity to the process by specifying targeted outcomes that the game must meet for it to be considered successful.

Motivational design in software applications is not new, with many underlying theories being researched and evaluated in this context (Rieber 1996). Cognitive load theory, for example, which evaluates the ability for an individual to process information and interactions, has been reviewed within multimedia learning environments (Kilic et al 2012). Other theories, including self-determination theory and Skinner’s behavior theory, have been evaluated in web-based learning software, preventive health care software, and online community software (Gurzick 2009, Matsuo 2008, Sundar 2012). Skinner’s behavior model has also been recognized as a means for effective results in wellness and health care applications, and is identifiable in effective systems like Weight Watchers weight loss programs (Freedman 2012).

Fogg’s Behavior Model, Behavior Grid and Persuasive Systems Design (PSD) are explored in terms of persuasive design of software systems, including behaviour change support systems, ecology, and more (Ferebee 2010, Fogg 2009, Oinas-Kukkonen 2010, Räisänen et al 2010). Other persuasive design theories draw from many theories of motivation and have been explored in areas such as interactive mannequins, exercise coaching, autism, and managing work-related stress (Harjumaa 2009, Ploug 2010, Ranfelt 2009, Reitberger 2009). Using the acronym OCEAN, VandenBerghe
proposes five elements of motivation that should be considered when developing games: openness, conscientiousness, extraversion, agreeableness, and neuroticism (VandenBerghe 2012).

Motivational theories have been an important part of the game design scene over the last 40 years (Baranowski et al. 2008, Engelfeldt-Nelson 2006, Malone 1980). Much of the research that has been performed has focused on evaluating games post-production against various theories and frameworks, including the Player Experience of Need Satisfaction (PENS), which is based on self-determination theory (Ryan et al. 2006). Considering these theories earlier in the design process is becoming more commonplace, with game designers from LucasFilm Games and WB Games recently describing at the 2013 Game Developers Conference how self-determination theory has drastically changed their approach to game design (Rocchia 2013).

One of the more interesting and potentially most impactful areas of motivational design in software is in the area of individual and community wellness (Yim et al. 2007). Social cognitive theory has been evaluated in light of serious games for health (Bandura 2004), but Bandura does not directly propose a method for use in game design. In this work, Bandura notes that:

_Human health is a social matter, not just an individual one. A comprehensive approach to health promotion also requires changing the practices of social systems that have widespread effects on human health…. Interactive computer-assisted feedback provides a convenient means for informing, enabling, motivating, and guiding people in their efforts to make lifestyle changes…. These systems need to be structured in ways that build motivational and self-management skills as well as guide habit changes. Otherwise, those who need the guidance most will use this tool least._ (p. 149)

Other research evaluates games post-production against theories of motivation and does not provide a mechanism for novice game designers to consider these theories when designing serious games for health.

3. Proposed Methodology

The author recently proposed a set of prompts for use in teaching motivation in serious game design based on Self-Determination Theory. The prompts were introduced near the end of the design phase, as development was already being undertaken. After testing these prompts with undergraduate research students, the author noticed that some students were more likely to defend the current design rather than take a deep look at how the design might be improved based on the prompts.

The author proposes that a more comprehensive approach to motivational design be explored and that a full set of prompts and instructional materials based on this design be created to teach motivational design in serious games. The most critical part of this will be the creation of a set of prompts that can be used as a tool for instructing novice game designers to consider player motivation within the game. By developing these resources, novice designers will be able to actively and fully consider the motivational aspects of gameplay in serious games to improve patient outcomes in the crucial early stages of design.

The author proposes to review Svinicki’s (2010) Combined Motivation Theory (CMT), a theory that was created as a framework for research in engineering education. CMT draws from six theories (self-determination theory, expectancy/value theory, behavior theory, social cognitive theory, attribution theory and achievement goal orientation theory), and has three basic pillars emphasizing the perceived value of the targeted task the learner’s beliefs about who controls the outcome and the learner’s self-efficacy for the task—that is, the belief in one’s own ability to succeed.

Svinicki postulates that the motivation of a learner is affected by (p. 16):

- The value of the targeted task, which is influenced by:
  - How interested the individual is in the task
  - How challenged the individual is
  - How influential others see it
  - The value that peers place on the task
• The individual’s beliefs about who controls the outcome
  o The amount of control given to the individual
  o The likelihood that the outcome can change given different circumstances
  o The ability to control the change in the outcome
• The individual’s self-efficacy
  o Past success at same or similar tasks
  o Having most skills required by the task
  o Being persuaded by someone else that success is possible
  o Seeing someone like themselves be successful
  o How difficult the individual perceives the task to be

Though these have been framed as learner needs, it is interesting to review these components as they apply to motivation in games, particular games for health which attempt engage the player in the game in order to promote behavior changes and/or learning.

The author proposes to review literature to define each of the theories in CMT, to evaluate how they have been previously applied to games, and to incorporate CMT in the creation of instructional materials. The instructional materials, including prompts, will also be created based on this theory and then tested with undergraduate students studying game design in Spring 2014.

4. Conclusion

The author proposes that a set of prompts rooted in various motivational theories would serve as a valuable resource for students in the early stages of game design. Though the intent of this research is to provide a tool based on motivational theories for the use by undergraduate students studying game design, the prompts may also serve as a resource for the larger design community. The research proposed in this abstract is the next logical step in establishing these prompts. For the purposes of the Psychology of Programming Interest Group (PPIG) Work-in-progress Workshop, the author seeks feedback on the proposed methodology.

5. References


Analysing the effect of Natural Language Descriptions as a support for novice programmers.

Edgar Cambranes, Judith Good, Benedict du Boulay
Human-Centred Technology Group
School of Engineering and Informatics.
University of Sussex.
Brighton, United Kingdom.
E.Cambranes-Martinez@sussex.ac.uk

Keywords: POP-I.A. learning to program; POP-II.A. novices; POP-II.B. debugging, program comprehension, design.

Abstract

This paper describes an experiment that has explored the effect to integrate natural language (NL) descriptions into a visual language (VL) designed for novice programmers. The experiment compared three conditions: the VL with NL representation, the VL with pseudo-code representation and the VL without second representation. The initial analysis measured the learning gain (LG) using the pre-test and post-test scores. Comparison of overall LG scores revealed no significant difference between the groups. However, the two groups with a secondary representation, and particularly the NL group, might be better assisted with some of their programming tasks.

1. Introduction

This research proposes to explore the usefulness of providing a second representation to support a visual language (based on flowcharts) designed for novice programmers. In particular, the second representation uses Natural Language to describe the current solution. The aim of the research is to assess the beneficial effects (if any) of a second representation on the comprehension, creation and debugging of programs. Specifically, the research will try to answer to what extent does the second representation affect these different aspects of the programming process?

My proposal considered a tool called Origami (Rodríguez-Cámara, Ku-Quintana, & Cambranes-Martínez, 2009), which is used to teach basic programming concepts to novices. This tool enables students to create, test and debug programs using a visual language based on flowcharts. The programs produced are syntactically error-free, however, students may input instructions into each flowchart block that are semantically incorrect. Origami logs interaction data during the problem solving process: time taken to complete an exercise, the blocks used, the inner blocks’ instructions and any compilation messages. Each interaction is associated with a timestamp that preserves the chronology of the problem solving process. The Origami tool was improved using some design ideas from Flip (Good, 2011) and now combines a visual language based on interlocking blocks with natural language

2. Study

Since the major consideration for this research is to examine the effect of providing secondary NL descriptions to support novice programmers, a comparative study was carried out with three conditions: the earlier version of Origami (with no secondary representation), a version with an secondary NL description, and a version with a secondary pseudo-code representation. This three-way comparison was an attempt to partial out the effects of providing a secondary representation of any kind from the particular effects of using NL in that role. Origami logs provide the main quantitative data source; whereas qualitative data has been gathered via semi-structured interviews.
The study took place in a public university in the south of Mexico. The participants were 75 first year students of a Computer Science 1st year course (63 males and 12 females), aged between 17 and 21 years old. Three groups were formed and were taught the same material by two tutors.

The materials used in the study were a programming pre-test and post-test, the Origami tool (three versions), and 3 sets of algorithmic exercises. Each group accessed all the materials and software through an online course delivery tool. The pre-test and post-test and each set of algorithmic exercises consisted of 3 types of exercise (completion, debugging and tracing) each with three difficulty levels (simple, medium and complex). The format of the items for the pre- and post-test were based on multiple choice questions (debugging and tracing) and open questions describing an algorithmic solution using natural language (program completion), and were administered via the online course delivery tool. The sets of algorithmic exercises were solved either using the Origami tool for work out of class or administered as online activities during normal lab sessions. This meant that some of the student work was unsupervised and unobserved except for the logs laid down by the system.

2.1 Initial Results

This initial analysis is based only on the pre- and post-test scores, and the derived learning gains (LG). There was no significant difference in LG between the three groups, though a trend in the data favoured the two groups with a secondary representation, and particularly the NL group. A more detailed analysis compared LG by types of exercise and by the ability level of the students. There was a trend in the data suggesting that the LG scores for Completion and Tracing exercises were greatest for the NL group.

<table>
<thead>
<tr>
<th></th>
<th>Overall</th>
<th>Completion</th>
<th>Tracing</th>
<th>Debugging</th>
<th>Top</th>
<th>Bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>0.2839</td>
<td>0.6601</td>
<td>0.5541</td>
<td>-0.0942</td>
<td>0.0449</td>
<td>1.3175</td>
</tr>
<tr>
<td>PC</td>
<td>0.0955</td>
<td>0.224</td>
<td>0.5069</td>
<td>-0.2112</td>
<td>-0.0638</td>
<td>0.7792</td>
</tr>
<tr>
<td>FC</td>
<td>0.1533</td>
<td>0.2361</td>
<td>0.3977</td>
<td>-0.08</td>
<td>0.0836</td>
<td>1.1381</td>
</tr>
</tbody>
</table>

Finally, when ability quartiles based on pre-test scores were analysed, there was a trend suggesting that LG for the bottom quartile were greater than for the top quartile.

3. Future Work

Further steps will include the open question answers: a qualitative analysis will be carried out using a similar coding scheme based on the developed in (Good, Howland & Nicholson, 2010). Additionally, the set of log data from the algorithmic exercises will be analysed.

4. References


Workshop: Psychological Impact of 3D Intelligent Environments on Pedagogy

Martin Loomes, Georgios Dafoulas and Noha Saleeb

Middlesex University, UK
m.loomes@mdx.c.uk, g.dafoulas@mdx.ac.uk, n.saleeb@mdx.ac.uk

Abstract

The ultimate goal of this research is to enhance students’ blended learning experiences comprising of both face-to-face and online courses. This is by investigating educational methods, factors, and procedures that can psychologically affect students to help achieve maximum engagement using Artificial Intelligence for e-learning within 3D Virtual Learning Environments (VLEs). The enhanced student engagement aspired for includes increased academic retention (assimilation), participation and overall enjoyment. 3D Virtual Learning Environments (3DVLEs) have been a host for many virtual campuses of universities, e.g. Harvard and Cambridge, since their offset more than a decade ago. These virtual media offer innovative opportunities for technologically supported pedagogy and e-learning for many fields of sciences and arts which has reaped noticeable participation, satisfaction and hence achievement from students. This supports the developmental perspective of teaching and transmission of knowledge by adopting the constructivist paradigm/approach to teaching and learning.

There is a challenge to integrate contributions from a number of different disciplines into a single learning support offering that will (i) take under consideration the psychological and pedagogic needs associated with the use of 3D VLEs, (ii) address usability and web 2.0 issues from the use of a social learning network and (iii) investigate 3D VLE interactions with the mediums used to access learning platforms. So far the creation of intelligent 3D VLEs is primarily concerned with the design of content for virtual learning tasks.

This workshop researches the opportunities available to use 3DVLEs, such as Second Life, to create e-learning Project Innovation for students using 3D Virtual and Generative Intelligent concepts. This entails the use of programming and coding to create bots (artificial intelligence robotic avatars) that can be used to direct interactive teaching and learning activities inside a 3DVLE, hence affecting the psychology of pedagogy offered within it. Moreover, through the creation and coding of holographic platforms (holodecks) inside 3DVLEs, diverse classroom and environment settings can be created to aid in the e-learning process and help the students themselves to use this technique to create immersive 3D projects e.g. 3D catalogues and exhibitions. This is in addition to the prospects of using these holodecks for educational role-play activities, modelling activities and interactive discussions and seminars.

The workshop session will consist of (i) an initial presentation explaining how the integration of virtual worlds and traditional physical environments lends an innovative approach to the enhancement of Higher Education Institutes provision, (ii) a strategic perspective on supporting teaching and learning via activities, assessment, feedback provision, pastoral services & student support, employer engagement, (iii) a hands on activity of the Middlesex virtual learning villages and, (iv) a brainstorming session on the future of blended learning using Artificial Intelligence, it’s psychological impact, and further institutional enhancement. Furthermore there are recommendations for possible applications of the technology through the use of a variety of educational scenarios in 3D Virtual Learning Environments. The key contribution is to initiate discussions, trigger debates and offer brainstorming opportunities related to enhancing the psychological effects of 3D Intelligent Environments on e-learning opportunities. The session is highly interactive and requires engagement in a series of activities, including hands on experience for all willing to access the Virtual Campus and participate in a 3D learning session.
Tools to Support Systematic Literature Reviews in Software Engineering: Protocol for a Feature Analysis

Christopher Marshall & Pearl Brereton
School of Computing and Mathematics
Keele University
Staffordshire, UK
c.marshall@keele.ac.uk / o.p.brereton@keele.ac.uk

Keywords: systematic literature review, automated tool, mapping study, software engineering, feature analysis

Extended Abstract

Systematic literature reviews (SLRs) have become an established methodology in software engineering research (Kitchenham et al., 2009). SLRs are concerned with rigorously evaluating empirical evidence from a variety of relevant literature, in an attempt to address a particular issue or topic of interest (Kitchenham & Charters, 2007). SLRs differ from ordinary literature reviews in being formally planned and methodically executed (Staples & Niazi, 2007). Undertaking an SLR comprises several discrete stages that can be grouped into three core phases: planning, conducting and reporting.

Despite their usefulness, conducting an SLR remains a highly manual, error prone and labour intensive process (Riaz et al., 2010; Babar & Zhang, 2009; Brereton et al., 2007). Automated support to assist the various stages of the SLR process may help to overcome some issues and so we are currently investigating the use and effectiveness of tools to support the conduct of SLRs in software engineering.

A literature review, in the form of a mapping study, identified tools that have been developed or used to provide automated support for the SLR process and established the extent to which the tools have been evaluated. The mapping study, after applying inclusion/exclusion criteria, accepted 14 papers into the final set. A variety of approaches and support tools which had been developed to assist the conduct of an SLR were found. During the next phase of the research project, using the Feature Analysis approach, four existing tools identified by the mapping study will be evaluated.

Feature Analysis is a qualitative form of evaluation involving a subjective assessment of the relative importance of different features and how well features are implemented (Kitchenham et al., 1997). It is based on the requirements that users have for a particular task/activity and mapping those requirements to features that a tool aimed at supporting the task/activity should possess. (Kitchenham & Jones, 1997; Kitchenham et al., 1997). Building on the results of the mapping study, a Feature Analysis protocol (plan) is being developed. Four SLR tools will be subject to the Feature Analysis.

These tools have been developed from scratch, with the sole aim of supporting software engineering researchers throughout the entirety of a SLRs undertaking. Table 1 lists the four candidate tools for evaluation.

The Feature Analysis will be the first step toward the development of a rigorous evaluation framework for tools that support SLRs. The aims of this activity are to produce an initial set of features and user requirements for a tool of this type. Results generated will act as a foundation for more complex evaluation exercises to be undertaken. The Feature Analysis will be organised as an initial screening, and will only focus on evaluating simple features. Simple features relate to aspects that are either present or absent and are assessed by a simple YES/PARTLY/NO nominal scale (Kitchenham et al., 1997).
A protocol for the Feature Analysis is currently being developed and will be presented at PPIG-WIP 2013. The presentation will include an initial feature list, score sheet and judgment scales and a discussion of future research aims and objectives.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SLuRp”</td>
<td>An open source web enabled database that supports the management of SLRs. (Bowes et al., 2012)</td>
</tr>
<tr>
<td>“SLR-Tool”</td>
<td>A freely-available tool for performing SLRs (Fernández-Sáez et al., 2010)</td>
</tr>
<tr>
<td>“StArt”</td>
<td>A tool that provides support to each of the activities, except the automated support of primary studies. (Hernandes et al., 2012)</td>
</tr>
<tr>
<td>“SLRTOOL”</td>
<td>An open-source, web-based, multi-user tool that supports the SLR process. (<a href="http://www.slrtool.org">http://www.slrtool.org</a>)</td>
</tr>
</tbody>
</table>

Table 1. SLR support tools selected for the Feature Analysis

References


Bringing Heterogeneous Computing to the End User

Gordon Inggs¹ & David B. Thomas¹, Wayne Luk²

¹ Circuits & Systems Research Group, Dept. of Electrical & Electronic Engineering, Imperial College London
[g.inggs1;d.thomas1]@imperial.ac.uk
² Custom Computing Research Group, Dept. of Computing, Imperial College London
w.luk@imperial.ac.uk

Heterogeneous Computers are computing platforms that are comprised of two or more architecturally distinct computing devices, such as Multicore Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Increasingly commodity computational systems are comprised of up to several of these platforms, as the power and memory walls continue to limit the capabilities of CPUs, and alternatives such as GPUs and FPGAs continue to improve in functionality and efficiency.

This new paradigm offers opportunities, but also poses challenges. In fact, it has been shown that applications can execute with better performance on a homogeneous architecture rather than a heterogeneous one, despite the heterogeneous configuration ostensibly having more computing resources [1].

We have identified three challenges in using Heterogeneous Computers effectively:

1. Predicting the runtime characteristics of applications tasks upon the constituent platforms of the system.
2. A better-than-homogeneous partitioning of tasks across the constituent platforms.
3. Balancing end user objectives in the context of a large design space.

We believe that we can address the first two questions through automatic means [3], using a domain-specific toolflow [4]. Furthermore, we argue that domain specific heterogeneous computing represents an opportunity for the end user. By providing exponentially more potential configurations, the user may then select one that more closely matches their desired outcomes for the system, one of the goals of software engineering practice. However, this opportunity may only be realised if that choice is presented to the end-user in a form which is meaningful [2].

![Figure 1. Pareto Interface Flow Chart](image)

We propose a two-stage input process to bring heterogeneous computing to the end user as is illustrated in Figure 1. Firstly, the end user describes the problem that they wish to solve in a
graphical manner specific to a particular application domain, and identifies the platforms upon which the application will be run. Secondly, the system executes a limited subset of the specified problem to create a representation of the projected Pareto optimal tradeoff between the domain specific metrics of interest, such as Latency and Accuracy in the context of Computational Finance. Then the end-user may select where on this Pareto frontier they wish the execution of their problem to lie, providing the configuration information to the system.

Figure 2. Optimal partitioning of example problem set

Figure 2 illustrates the key element that the end user uses to explore the design space as part of this approach - a Pareto curve-based interface for a small computational finance application of three option pricing tasks upon two platforms, an AMD Opteron Multicore CPU and AMD FirePro GPU. The purple line plotted illustrates the trade-off curve that the end-user would have to select from in terms of absolute latency of the computation and minimum guaranteed accuracy. Helping make the case for heterogeneous computing, the heterogeneous implementation outperforms the homogeneous GPU implementation by 1.5 times, and the CPU one by 2.3 times, while delivering equivalent accuracy.

Further work will focus on an exhaustive elaboration of the design space of the Kaiserslatuarn benchmark of Option Pricing tasks upon a range of platforms. The aim of these experiments is to verify the efficacy of the prediction and partitioning infrastructure to use. Furthermore such an elaboration will allow the benefit of greater user choice versus a balanced default options to be quantified.

References


¹ http://www.uni-kl.de/en/benchmarking/option-pricing/
Integrating Connectivist Pedagogy with Context Aware Interactions for the Interlinked Learning of Debugging and Programming

Xiaokun Zhang

School of Computing & Information Systems, Athabasca University, Alberta, Canada
xiaokunz@athabascau.ca

Extended Abstract

Debugging and programming are important skills that remain to be challenge for beginner to learn and for instructors to teach. Research relating debugging ability to psychological factors or cognitive abilities has gained some successes, such as researches in learning styles [Goold & Rimmer 2000, Thomas, etc. 2002], visualisations enhanced multiple external representations [Pablo Romero, etc. 2008], visual-based independence tests (Mancy & Reid, 2004), and abstraction ability (Bennedsen & Caspersen, 2006). One of the recent works in Collabode system developed in MIT [Max Goldman and Robert C. Miller 2012] demonstrates an approach of combining test-driven pair programming with micro-outsourcing mechanism to leverage error-mediated integration of pairs of both student and professional programmers. It appears that explicit relations between debugging success and individual factors [Rene´e McCauley, etc. 2008] within emergent interlinked social learning context need further investigation, even though social cognition implications have been observed in relevant researches in terms of pair programming, collaborative and heterogeneous knowledge representation and sharing between various granularity levels in debugging processes. On the one hand, social cognition enhanced learning methods have been gaining innovative success in learning paradigms by means of fast growing social computing and context-aware collaborative computing technology, that reveals opportunity to foster social intelligence enhanced learning of debugging and programming. On the other hand, it is non-trivial to ensure high quality services of such interlinked learning from various perspectives especially how innovative learning theory and pedagogy methods are engaged in the practical and domain dependent application framework in order to facilitate personalized, social and context aware learning and teaching in heterogeneous learning environments, especially in the situation in which students, teachers, and learning resources and services are geographically located in many different locations. To this end, this paper introduces our on-going project that aims at creating an integrated interlinked learning context model under which emergent connectivist pedagogy concept is embodied by three key technique components: hypergraph-based relation data model for representing interlinked learning context; functions of observing and facilitating teaching and learning activities (data or actions) in connectivist learning; and loosely coupled collaboration mechanism for imposing individual interlink context towards self-organized problem-solving group structures by dynamic changes to temporal or spatial proximity suggested by the former two components.

In this paper, interlinked learning context and domain knowledge of debugging and programming are selected as one of application frameworks to demonstrate how proposed interlink context model and relevant data representation can leverage multifaceted interlink analysis for fine-grained data-driven context observation and event-driven loosely-coupled learning management through heterogeneous interaction processes. The paper may not have enough space to detail algorithm description and analysis in terms of interlinked learning context in debugging and programming case, rather, the paper describes hypergraph-based interlink context model and explains how tensor method is applied to model and analyse multifaceted interlink context to uncover the underlying debugging problem structures, students interaction structure and behaviours in debugging by means of known types of knowledge and debugging strategies in connectivist learning perspective. The generic concept model of multifaceted interlink learning context model and multilevel collaboration mechanism based on data driven interlink observation and event-driven interaction engagement in terms of teaching presence (TP), cognition presence (CP), and social presence (SP) in connectivist learning perspective,
is shown in Figure 1. The connectivist pedagogy paradigm demonstrated in Figure 1 has multi-level granularity in terms of collaborative information framework for facilitating heterogeneous interactions and interlinked learning activities based on macro-level problem description, macro-level interaction structure description, micro-level problem observation, and micro-level interlinked learning status and activity structure in dealing with fine-grained learning tasks in debugging, concept components in programming, and learner generated contents and reflections. The paper addresses key issues on how heterogeneous interactions between those parts from various granularity levels are collaboratively observed and analyzed in practical way in terms of the concepts of cognitive present, teaching present, and social present within connectivist learning strategies [George Siemens 2005, Terry Anderson and Jon Dron 2011, Athabasca University in Canada] in order to manage and optimize interlinked learning and teaching activities in open and dynamic social learning environments, especially in the scenario of the learning of debugging and programming.

It is emphasized that my colleagues Professor Siemens initiated the concept of connectivist pedagogy in 2005 and Professors Anderson and Dron enriched the theory by embracing emergent social computing features. However, practical interlinked learning system and implementation need not only high level pedagogy and conceptual interaction mechanism, but also measurable, retrievable, and observable data representation and algorithm mechanism under certain collaborative framework and architecture to leverage appropriate learning analysis and management. The project demonstrated in this paper addresses these issues from implementation perspective and focuses on specific elearning system in the learning of debugging and programming.

Figure 1. Multi-faceted interlink learning context model and multilevel collaboration mechanism based data driven interlink observation and event-driven interaction in terms of teaching presence (TP), cognition presence (CP), and social presence (SP) in connectivist learning perspective