6th Baltic Sea Conference on Computing Education Research

Koli Calling 2006

Anders Berglund and Mattias Wiggberg (Eds.)

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2007-006
February 2007
ISSN 1404-3203

UPPSALA
UNIVERSITET

6th Baltic Sea Conference on Computing Education Research
Koli Calling 2006

BY
ANDERS BERGLUND
MATTIAS WIGGBERG
EDITORS

February 2007

DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

6th Baltic Sea Conference on Computing Education Research
Koli Calling 2006

Anders Berglund
Anders.Berglund@it.uu.se

Mattias Wiggberg
Mattias.Wiggberg@it.uu.se

Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-3203
Printer Uppsala University, Sweden

Foreword

You are holding in your hands the proceedings of the 6th Baltic Sea Conference on Computing Education Research - Koli Calling. The conference was held in November 2006.

The papers presented at the conference, and collected in these proceedings were of excellent quality and highlight both the depth and the variety of the emerging field of Computing Education Research.

Twenty-nine papers/posters, one invited seminar and two invited speeches were presented during the three conference days. Lecia Barker, ATLAS, University of Colorado, Boulder, CO, USA, broadened our perspective on students' experience of power and gender, through her speech *Defensive climate in the computer science classroom*, originally written by Dr Barker together with Kathy Garvin-Doxas, and Michele Jackson for the 33rd SIGCSE technical symposium on Computer science education, 2002. The Koli Calling conference thanks ACM for a travel grant for Dr Barker. Tony Clear took the discussion further, by introducing critical enquiry to the community in his talk *Valuing Computer Science Education Research?*, finally Matti Terdre discussed the interaction between computer science and the community in his seminar *The Development of Computer Science: A Sociocultural Perspective.*

Contributions to the main conference can take one of three forms. *Research* papers present unpublished, original research, presenting novel results, methods, tools, or interpretations that contribute to solid, theoretically anchored research. *System* papers describe tools for learning or instruction in computing education, motivated by the didactic needs of teaching computing, while Discussion papers provide a forum for presentation of novel ideas and prototypes. Finally, the *Posters* had their focus on work in process. All papers and posters were double-blind peer reviewed by members of the international program committee.

Lively discussions are a hallmark of the Koli Calling conference, with the whole conference coming to serve as a forum for a development of new ideas. This is a tradition from the previous years, that was followed this year. Maybe it became even stronger, as the Speaker's Corner, turned out to be a useful discussion area.

The conference was organized by the UpCERG group, Department of Information Technology, Uppsala University, Sweden, in collaboration with ACM SIGCSE, USA and CeTUSS, Centrum för Teknikutbildning i Studenternas Sammanhang, Uppsala University, Sweden. The practical arrangements were made by the Department of Computer Science, University of Joensuu, Finland.

The success of the conference is to a large degree due to the open and friendly atmosphere that encourages the participants to return to this Finnish site. Many of the around 40 participants from nine countries on three continents had attended the conference during earlier years. As programme chair, I would like to thank everyone who made this development possible. Particularly, I want to mention the programme committee for their dedication to providing constructive criticism of the submissions, and the organising committee who all worked very hard to make the conference a success.

However, most important is the contributions of the authors and the participants, without whom we would not have had a conference.

When citing papers at this conference, please use the following format:
<Author(s)>. (2007). <Title>. In A. Berglund & M. Wiggberg (Eds.) *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling*. Uppsala University, Uppsala, Sweden. Also available at http://cs.joensuu.fi/kolistelut/

I look forward to the next Koli Calling Conference, 2007.

Uppsala, February 2007,

Anders Berglund

Conference Chair

Anders Berglund	Uppsala University, Sweden

Programme Committee

Michael E. Caspersen	University of Aarhus, Denmark
Valentina Dagiene	Vilnius University, Lithuania
Mike Joy	University of Warwick, UK
Ari Korhonen	Helsinki University of Technology, Finland
Raymond Lister	University of Technology Sydney, Australia
Lauri Malmi	Helsinki University of Technology, Finland
Arnold Pears	Uppsala University, Sweden
Guido Roessling	Darmstadt University of Technology, Germany
Tapio Salakoski	University of Turku, Finland
Carsten Shulte	Freie Universität Berlin, Germany
Jarkko Suhonen	University of Joensuu, Finland
Erkki Sutinen	University of Joensuu, Finland

Organizing Committee

Tomi Mäntylä	University of Turku, Finland
Jussi Nuutinen	University of Joensuu, Finland
Marjo Virnes	University of Joensuu, Finland
Mattias Wiggberg	Uppsala University, Sweden

Contents

Discussion Papers

Demo/Poster Papers

Invited Speakers

Defensive Climate in the Computer Science Classroom

Lecia Jane Barker
University of Colorado
Boulder, CO 80309 USA
Lecia.Barker@Colorado.edu

Kathy Garvin-Doxas
University of Colorado
Boulder, CO 80309 USA
Ronda.Garvin-Doxas@Colorado.edu

Michele Jackson
University of Colorado
Boulder, CO 80309 USA
Michele.Jackson@Colorado.edu

ABSTRACT

As part of an NSF-funded IT Workforce grant, the authors conducted ethnographic research to provide deep understanding of the learning environment of computer science classrooms. Categories emerging from data analysis included impersonal environment and guarded behavior, and the creation and maintenance of informal hierarchy resulting in competitive behaviors. These communication patterns lead to a defensive climate, characterized by competitiveness rather cooperation, judgments about others, superiority, and neutrality rather than empathy. The authors identify particular and recognizable types of discourse, which, when prevalent in a classroom, can preclude the development of a collaborative and supportive learning environment.

.

Valuing Computer Science Education Research?

Tony Clear
Auckland University of Technology
Private Bag 92006, Auckland
New Zealand
tony.clear@aut.ac.nz

ABSTRACT
This paper critically enquires into the value systems which rule the activities of teaching and research. This critique is intended to demonstrate the application of critical enquiry in Computer Science Education Research and therefore uses critical theory as a method of analysis.

A framework of *Research as a Discourse* is applied to explore how the notions of research as opposed to teaching are presented, and how discipline and research communities are sustained. The concept of a *discourse*, based upon the work of Foucault, enables critical insight into the processes which regulate forms of thought.

This paper positions the field of Computer Science Education Research, as an illustrative case, within the broader discourse of *Research*, and argues that Computer Science Education Researchers and educators need to understand and engage in this discourse and shape it to their own ends.

Keywords
CS Ed Research, Critical Theory, Post-Modernism, Discourse Analysis, Research Assessment, Research Quality

1. INTRODUCTION
This paper reviews the modern perspectives framing the notion of 'research', and the manner in which teaching and research in the academy are often juxtaposed in a false dichotomy, wherein teaching practice is very much the poor cousin.

A critical framework of 'research as a discourse' is introduced and then applied, in order to enquire into the powerfully reinforced value systems which rule the lives of academics in computer science (among other disciplines). Computer science education research (CS Ed research) itself is scrutinized as one illustrative case of such discourse. CS Ed researchers and educators are urged to be conscious both of the contexts within which they operate, and these sets of broader shaping forces. Armed with this knowledge then, CS Educators can become more proficient both within their practice and their research. The paper concludes with recommendations by which CS Ed researchers might shape these discourses to their own ends, in furthering CS Ed research and their own professional teaching practice.

2. RESEARCH AND SCHOLARSHIP
The notion of "research' has acquired a particular set of meanings in today's academy. As Lévy-Leblond has observed, it is only in this century that forms of specialisation in academic work have evolved, echoing the "specialisation, fragmentation and hierarchisation" [33] of industrial work. He asserts therefore that, "The word 'researcher' is quite new; in the past there were only "scholars", whose activity consisted not only in doing research, but also in teaching, disseminating and applying science" [33].

Talking more specifically of computer scientists Ray Lister [35] opines, that we lead double lives, engaging actively with a community of colleagues in our "outward looking" research lives. But in contrast "our teaching lives are inward looking. We may talk to our colleagues about teaching, but in those conversations we regard introspection and "gut feel" as legitimate justifications of our beliefs" [35]. This of course differs from the rigour applied to our research lives, wherein we build upon prior research cycles.

While Lévy-Leblond may lament the modern triumph of research over scholarship within academia, Ray Lister laments the lack of scholarship frequently applied in our teaching lives. Both observations demonstrate a problematic dichotomy, in which a dynamic teaching-research nexus is conspicuously absent.

2.1 Beyond the Dichotomy
Boyer [11] moves beyond the narrow research-teaching distinction in proposing four forms of scholarship which cover the dimensions of a University educator's job, namely: the scholarship of discovery; the scholarship of integration; the scholarship of application; and the scholarship of teaching.

For Boyer the scholarship of *discovery* is what is typically meant when academics speak of 'research' [11]. Central to higher learning is the commitment to "knowledge for its own sake, to freedom of inquiry and to following in a disciplined fashion, an investigation wherever it may lead" [11].

The scholarship of *integration* involves transcending the restrictions of discipline boundaries. Akin to the scholarship of discovery, it involves research at the boundaries where fields converge. It seeks new combinations of fields "as traditional disciplinary categories prove confining" [11]. CS Ed research, as a transdisciplinary endeavor, can be seen to reside within this category.

The third form of scholarship, the *application* of knowledge, involves professional activity based upon a field of knowledge. This is an interactive form of scholarship, occurring in professional contexts such as medicine and Information Technology wherein theory and practice interact and inform one another. Therefore it rejects the linear view that knowledge must first be discovered before being applied.

The fourth form of scholarship is the scholarship of *teaching*, the role of which for Boyer is to both educate and entice future scholars. Teaching creates a common ground of intellectual commitment, stimulates active not passive learning and

encourages students to be critical, creative thinkers, and lifelong learners. "Further, good teaching means that faculty, as scholars, are also learners" [11].

CS Ed researchers need to apply a judicious mix of these four forms of scholarship. In a cyclical model, discipline originated topics and concepts may be developed and refined in our teaching context, and in turn informed through the scholarship of integration by CS Ed research programs which systematically evaluate the effectiveness of our interventions.

2.2 Research – Definition and Measurement

A political perspective on how research is shaped posits that "scholarly endeavours are ultimately defined by the interest of those who dominate society and by whose largesse academics retain the privilege of pursuing research…The interests of the powerful are said to shape research more significantly than the curiosity of the researcher, primarily because the former control the latter's access to critical resources" [9]. Definitions of research and the way in which it is measured and rewarded are crucial mechanisms for regulating behaviour and directing resources in governmental, commercial and academic domains.

So how do these influential 'patrons' view research? How is it defined, and how are outcomes measured?

One such patron is the OECD, (the European umbrella group for Economic Co-Operation and Development).

A highly influential OECD report (the Frascati manual), contends that it is: "a cornerstone of OECD efforts to increase the understanding of the role played by science and technology by analysing national systems of innovation…providing internationally accepted definitions of R&D and classifications of its component activities" [40, p.3]. The report further claims to have become "a standard for R&D surveys worldwide".

In the OECD definition research and experimental development comprise, "creative work undertaken on a systematic basis in order to increase the stock of knowledge, including knowledge of man, culture and society, and the use of this stock of knowledge to devise new applications" [40, p.30].

The manual in addition, explicitly defines what is ***not*** to be regarded as research. For instance, "All education and training of personnel in the natural sciences, engineering, medicine, agriculture, the social sciences and the humanities in universities and special institutions of higher and post-secondary education should be ***excluded***" [40, p.31].

Yet some forms of education (doctoral level study and supervision activities) are decreed to be a 'borderline area' for inclusion as R&D. The answer appears dependent upon the degree to which the study and supervision activities contain a sufficient element of novelty and have as their object to produce new knowledge.

The manual provides a further breakdown of R&D, into three categories:

- **basic research** (without any particular application or use in view);
- **applied research** (directed primarily towards a specific practical aim or objective) and
- **experimental development** (directed to producing new materials, products or devices, to installing new processes, systems and services, or to improving substantially those already produced or installed).

These definitions of Research from the Frascati manual then, largely map to Boyer's scholarship of *discovery*, (and may extend to that of *integration)*. *Experimental development* on the other hand reflects the scholarship of *application* and maps closely to the process of research commercialization, in a classic linear model of scientific discovery, technology development and subsequent commercialization.

2.2.1 An Economic Lens

Here we see highlighted the economic lens through which the whole endeavour of research is viewed. This is a natural enough perspective from the OECD, which uses the definition to create a basis for comparable national statistical measurements of R&D efforts. However we see a similar policy perspective on research espoused more recently by the ACM Job Migration Task Force:

"For a country to have companies that are at the forefront of innovation is generally seen as essential for robust economic growth in the long term. …Fostering research…creates cutting edge technology and it hones the skills of cutting edge personnel. The importance of research in and of itself is demonstrated by figure 14 which shows nine industries, each worth at least a billion dollars, spawned by IT research…The main point is that research is a driver of major economic development, and government funding has historically played an important role in priming these developments" [6, p. 175-6].

Thus it can be seen that research is often seen instrumentally by policy makers, governments and private patrons of research projects. Whether at a project level or at a country strategic level, research is viewed as a competitive investment with the hope of gaining a return. Governments frequently invest in research indirectly through general funding to universities for education and research, where "Such flows may represent up to over half of all support for university research and are an important share of all public support for R&D" [40, p.21]. Therefore governments have a legitimate interest in mechanisms both to allocate and to account for the effectiveness of these significant public investments.

2.2.2 Measuring Research - Impact on Educators

This need has seen several models of research assessment being applied. A review of international research assessment practices [51] has identified "4 categories of countries in regards to university research funding practices". The first group of countries used a performance based approach to distribute funds; the second used an indicator other than research evaluation, such as student numbers; the third group in which research allocations were 'open to negotiation'; the fourth where research assessment and funding were separated.

Performance based schemes (e.g. the UK RAE and New Zealand PBRF) attempt to assess the quality and quantity of research being produced, in order to determine funding for each institution. The definition of research applied in these schemes determines what forms of research will be encouraged and valued. Interestingly the New Zealand Performance Based Research Fund (PBRF) definition of research borrows heavily from the OECD definitions, and again explicitly excludes "preparation for teaching" [2]. As observed in a review of the PBRF impact on the subject of education, "some will claim that the PBRF definition of research excludes many activities and outputs central to the discipline of education" [2]. Similarly in the review of the UK's RAE exercise, respondents argued that the RAE has "neglected pedagogical research by 'hiving it off' to the education panel for consideration, rather than assessing it

within its parent subject panel" [44]. A further issue noted was the encouragement of an undue focus by academics on research rather than teaching, which was "perceived to have driven wedges between teaching and research" [44].

In the New Zealand context, the review conducted by [2] concluded that education was one of the poorest performing discipline areas in the research performance exercise, with some 73.7% of the nation's education academic staff being deemed to be 'research inactive' (or in other words they failed to meet the threshold required for their research to even rate within the system). This could be partly explained by the dual system of professional colleges of teacher education and universities, with the professional colleges' results showing 90.7% of their academics to be so-called 'research inactive', as opposed to the universities with 54% 'research inactive'. More positively in the New Zealand context, the PBRF subject panel of Mathematical and Information Sciences explicitly defined its subject area to also include "pedagogical research in computer and information systems" [49, p. 116].

It was also suggested that the more practical forms of curriculum advice and classroom teacher support provided by professional teacher educators, failed to result in findings which were "open to scrutiny and formal evaluation by others" [2] with peer review being "the litmus test of what is and what is not research for PBRF purposes" [2].

Such poor outcomes for the education discipline demonstrate the inherent bias against education and pedagogical activities underpinning such research measurement schemes.

Referring back to the Frascati manual then, we can see the underpinning utilitarian mindset in the mental model that ***education*** is merely the ***transfer*** of existing knowledge. This contrasts poorly with the view of ***research***, as the ***creation*** of new and potentially wealth-creating scientific discoveries as implied by the scholarship of discovery.

3. CRITICAL PERSPECTIVES AND THE NATURE OF KNOWLEDGE

Space precludes a full elaboration here of the nature of critical theory. Interested readers are referred to [15] for further reading. Suffice it to say that critical research involves research based not upon the *natural sciences*, or the *interpretive sciences*, but upon the *critical sciences* as distinguished by Habermas [28].

In such a model of research the researcher directly addresses issues to do with power, distortions of communication and the ways in which power structures are created and sustained. The ***critical*** method has an explicitly ***emancipatory*** mission, with an interest in addressing issues to do with power imbalances and liberation from unwarranted forms of constraint. This paper will attempt to expose to scrutiny the role of 'discourse' in shaping the lives of CS educators and CS Ed researchers, in the hope that by a greater awareness of the forces shaping our activities we may be more effective in our education and our research.

Research and knowledge-seeking are inseparable. This is especially true with research in the critical paradigm where the very nature of knowledge is not assumed within the paradigm. Yet the very term 'knowledge' is an elusive notion. Michel Foucault, the French social historian and critical philosopher, discussed the concept of knowledge as a linked word structure. He refers to the concept as ***power/knowledge***, seeing the two as indistinguishable.

Foucault's argument is that "Knowledge and power are integrated with one another…It is not possible for power to be exercised without knowledge, it is impossible for knowledge not to engender power" [23].

This distinction can be readily illustrated by looking at the academic or professional disciplines. As Foucault points out, disciplines of their very nature are limiting. "…disciplines are defined by groups of objects, methods, their corpus of propositions considered to be true, the interplay of rules and definitions, of techniques and tools; all these constitute a sort of anonymous system, freely available to whoever wishes or whoever is able to make use of them, without there being any question of their meaning or their validity being derived from whoever happened to invent them" [24]. Moreover, for Foucault "A discipline is not the sum total of all the truths that may be uttered concerning something" [24].

Therefore, just as "medicine does not consist of all that may truly be said about disease" [24], likewise today it is true that each of the sub-disciplines of computer engineering, computer science, software engineering, information technology, information systems [cf. 47], does not represent the sum total of all the truths one could say about computing.

4. RESEARCH AS A DISCOURSE

To discuss research then, in applying a critical perspective from Foucault, these concepts of *power/knowledge* and *the disciplines* are fundamental. A further important concept from Foucault is that of a *discourse*.

A discourse is a "regulated system of statements and practices that defines social interaction. The rules that govern a discourse operate through language and social interaction to specify the boundaries of what can be said in a given context, and which actors within that discourse may legitimately speak or act" [17].

The whole topic of *research*, and its subset *CS Ed Research*, fits within such a definition. It is also a discourse distinct from that of CS education, which has its own discourse structures.

The principles by which discourse is regulated have been identified in table 1 below as: exclusion, limitation and communication. These could be rephrased as: 1) what is not said; 2) what may not be said and 3) how things may be said.

Table 1. Principles of Discourse Regulation [excerpt from 17]

EXCLUSION		
Prohibition	**Division**	**Truth Power**
Taboos	Legitimate participation	True vs. false
LIMITATION		
Commentary	**Rarefaction**	**Disciplines**
Meaning rules maintained	Identity rules maintained	Belief rules maintained
COMMUNICATION		
Societies of Discourses	**Social appropriation**	**Systems of regulation and control**
Social group	Maintain or modify	Production and manipulation

In table 1 above the framework for discourse analysis from [17], is outlined. While originally applied to investigate the role of IT in organizational change, it is applied here to the rather different topic of research, as a means of understanding how "research" represents a constraining discourse for the CS Ed research community.

4.1 The Research Discourse - Forms of Regulation

Particular forms of regulation can be said to apply in the research domain. These are illustrated below with examples indicating how the discourse is sustained in practice, both for research in general and for CS Ed research in particular. In Foucault's words, "truth is a thing of this world: it is produced only by virtue of multiple forms of constraint" [23].

4.1.1 Exclusion – What is not said

4.1.1.1 Prohibition - research taboos?

Let us consider some typical taboos for a researcher:

- Lack of rigour or system in approach. Yet some researchers challenge the idea that a systematic approach based upon logic is the sole form of rigour, because of the inherent absence of a holistic view of the person which encompasses human essence and spirituality, as lived in community. Research with indigenous peoples often encounters these issues [cf. 10]. Heshusius [29] suggests "any concept of rigor related to participatory consciousness must not override the recognition of kinship and the centrality of tacit and somatic ways of knowing." Carter [12] argues for CS and Educational research using psychodrama and action methods to embody "the spontaneity and creativity of groups in the here and now".
- Use of emotion and subjectivity in writing (anathema to the natural sciences research paradigm, to which most CS researchers are accustomed). Yet Heshusius [29], critiques the whole distinction between objectivity and subjectivity, and the ability of researchers to manage and distinguish between their subjective (bad) and objective (good) selves? "Don't we reach out (whether we are aware of it or not) to what we want to know with all of ourselves, because we can't do anything else?"
- Unethical behaviour as a researcher, which brings with it an apparatus of ethics committees such as the Auckland University of Technology Ethics Committee, which has weighty sets of guidelines and formalised processes for critique of research proposals. Yet Zeni [52] asserts that most educational action research should be exempt from formal ethical review processes, and urges "academic institutions to support reflective teaching and to minimise the bureaucratic hurdles that discourage research by teachers to improve their own practice".
- Plagiarism, which brings complex and onerous rules and procedures for citation of previous researchers work.

4.1.1.2 Division - who may participate?

The principle of division operates to restrict who may be involved in research

- The type of educational institution may restrict the degree of involvement in research. More teaching intensive institutions will, by their very workload models, preclude the level and types of research that may be undertaken [16].
- The type of job classification may restrict the degree of involvement in research. For instance in [20] in the teaching intensive category of 'regular part time' faculty member "Scholarship is expected, but is often extended to include pedagogical as well as basic research".
- In New Zealand the PBRF limits its census of researchers to include only those staff teaching on degree programmes, who are expected to have higher qualifications. Gal-Ezer and Harel for instance assert that "it is reasonably obvious that college level teachers must be equipped with a doctoral degree in CS" [25]. Yet the nature of computing as an applied discipline means that competence as an educator may come from an industry background and lower qualification levels. In the US the ABET accreditation criteria for CS programmes acknowledge this with faculty accreditation criteria [1] which require only that "*some faculty* have a terminal degree in computer science" (CS). The general criteria require that "Each has a level of competence that normally would be obtained through graduate work in the discipline, *relevant experience*, or relevant scholarship" [1, p.21].
- The status and title of 'researcher' is certainly not granted to practitioners, who are considered to engage in 'routine' work. However the Frascati manual does acknowledge [40, p.46] that "The nature of software development is such as to make identifying its R&D component, if any, difficult" and in effect acknowledges that practitioners in this field are frequently researchers, since software development "may be classified as R&D if it embodies scientific and/or technological advances that result in an increase in the stock of knowledge" [40, p. 46].
- Experience in supervising postgraduate research is normally demanded in the profile for teachers on postgraduate programmes. These rules tend to marginalise those with a predominantly undergraduate teaching or even significantly senior practitioner background.

4.1.1.3 Truth power

"The principle of truth power occurs through creating opposition between the true and the false" [17]

- In the natural sciences tradition the classical scientific research paradigm and the techniques that accompany it, hypotheses, experimental designs, published outcomes etc. are the means to determining the validity of truth claims.
- The process of publishing whether in academic journal articles or in books is also a means of according research work the status of acceptable truth.
- A whole panoply of research outputs is defined through performance based assessment regimes [e.g. 49] and through journal selection and ranking exercises such as [42]. These serve to indicate the status and significance to be accorded different pieces of work.

4.1.2 Limitation – What may not be said

The principles of limitation "operate to classify, order and distribute the discourse to allow for and deal with irruption and unpredictability" [17].

4.1.2.1 Commentary -

Commentary "prevents the unexpected from entering into a discourse, [by maintaining the meaning rules and ensuring that] the new is based upon a repetition of the old" [17] through mechanisms such as:

- maintaining the paradigm, for instance CS debates concerning programming as "1) a manipulative tool for the conduct of algorithmic thought experiments in a purely scientific CS model, as opposed to 2) the centrality of design in the construction of large scale software systems by professional software engineers" [36].
- maintaining or attempting to define restrictively the discipline boundaries (eg. Computer Engineering, Computer Science, Software Engineering, Information Systems) [47]
- maintaining the disciplinary focus of research topics and issues of interest. For instance Ramesh et al., [42] in their study of selected computer science journals, observed that four primary research approaches have been applied – descriptive, developmental, formulative and evaluative. In their findings they noted that "the focus in most areas of computer science research is primarily on formulating things" and moreover "the two categories societal concepts and disciplinary issues are not represented at all" [42]. Given that the category of 'disciplinary issues' here includes "computing research" and "computing curriculum/teaching", this is a discouraging finding for those with an interest in computer science education research.
- As a relevant contrast from the Information Systems discipline, Liegle & Johnson, [34] report that of 61 top ranked IS journals only two declared a pedagogical focus, and less than 6% of the articles had a pedagogical focus. The top three journals were even less interested, with "an insignificant number of pedagogical articles".
- Of incidental interest with respect to this paper, the above study [42] found no articles applying the critical-evaluative research approach, for which this paper furnishes an example.

4.1.2.2 Rarefaction - identity rules maintained for members of the discourse community

"The principle regulates the discourse through the speaker conventions which prescribe the role of a speaker rather than an individual" [17].

- Research dictates prescribed ways of speaking, and roles for the researcher - normally that of "expert commentator"
- conference presentations are one formalised mechanism for maintaining the identity of speakers within the community. The roles of keynote speaker, invited speaker, paper presenter, poster presenter, session chair etc. are all prescribed roles within the research conference setting which reinforce status and validity of contribution

4.1.2.3 Disciplines

The disciplines limit the discourse "through the application of rules, definition, techniques and media" [17]. Mechanisms such as the following act to preserve the boundaries of disciplines:

- Maintaining definition of the discipline and techniques for its study. Much has been written about the Computer Science and Information Systems disciplines, to define them, give them status, attract resources to those engaged in researching in these fields, and moreover to define what they are not. For examples of computing discipline and curriculum discussions and proposals refer [47, 8, 13, 39, 18, 14].
- The world view of Computer Science which developed from the mathematical and scientific research communities, tends to be a largely objectivist one based upon the natural sciences. Information Systems developed from a hybrid background with a business, management and organization science perspective, is more accepting of research based upon the interpretive and critical sciences.
- Clark [14] positions Computer Science within a set of discipline dimensions as a "hard-applied" discipline; 'hard' in the sense of "having a body of theory to which all members of the discipline community subscribe", and "applied" in the sense of being "concerned with practical problems". However there are those computer scientists who see CS less as an applied engineering discipline than as a mathematical 'pure' discipline, concerned with universals. By contrast Clark contends that Education, as an area in which "content and method tend to be idiosyncratic" is a "soft-applied" discipline. The CS Ed combination then, will need to borrow from both discipline perspectives.
- Software engineering offers an interesting case study in discipline formation. In spite of pressure to professionally license software engineers to ensure public safety when developing safety critical systems, a task force established by ACM to review the proposals "concluded that licensing within the framework of the existing PE mechanism would not be practical or effective in protecting the public and might even have serious negative consequences" [31]. Some of this debate reflected the essential distinctions between engineering and computer science disciplines.
- A further role of the disciplines is to constrain the discourse within certain boundaries. Therefore they are inherently not trans-disciplinary, and tend to be restrictive of the scholarship of integration. Fortunately for educators in the computing field, there is a developing sub-discipline of CS Ed research [21, 41], which is congruent with many aspects of the practicing CS educator's computing discipline focus. CS Ed practitioners can become researchers through contribution to the many journals and conferences offering opportunities to present work in this area. The proposal by Seidman et al., [46] on maintaining a core literature, in itself represents a further initiative to define the discipline of CS Ed Research.

4.1.3 Communication – How things may be said

The principles of communication concern the "conditions in which communication is conducted, including the ritual framework surrounding all discourses." [17].

4.1.3.1 Societies of discourse

This principle operates to restrict communication to those who are a member of certain social groups, such as members of a discipline community. The principle operates by:

- Restricting research to the academic community and postgraduate scholars, or those commercial researchers who have acquired funding from some source
- Delegitimising educational practitioners as researchers, since as noted in section 2.2 above the process of education/course development etc. in itself is excluded from the definition of research [40, p31]. In a volatile field such as computing, the process of developing and delivering a new course may involve considerable research activity and scholarship and the course itself may represent new knowledge. Certainly the scholarships of integration, application and teaching are all involved.

- Delegitimising former practitioners turned educators. Their lack of formal credentialisation, such as Doctoral qualifications can serve to exclude them from research opportunities, funding for projects, promotions or acceptance as credible researchers. For instance the status of "Professor" carries considerable reputational value, but this rank is largely unachievable by those without doctoral qualifications.
- Failure to use formally prescribed methods, indicative of such rigour. Lay comment, insight or writings for instance are not deemed research. In the NZ research performance assessment exercise, Alcorn et al. [2] noted that educational researchers submitted ineligible items as their nominated research outputs for research assessment including; powerpoint presentations; textbooks where the research dimension was not apparent; papers submitted for postgraduate courses; production of material related to curriculum development workshops.
- The practitioner communities have their own societies of discourse - eg. User groups and professional forums. Some of these professional forums (eg. ACM, IEEE) are a meeting ground for both research and practitioner communities.

4.1.3.2 Social Appropriation

This principle serves to maintain or modify the discourse, through principles of communication that regulate and control membership of a discourse. The principle operates through:

- Prescribed forms of communicating research (language, style, methods - eg. experiments, use of statistical techniques such as analysis of variance (ANOVA) - research journals etc.) These prescribed norms serve to exclude the uninitiated.
- Editors, editorial policy, focus & philosophy of publication. Ramesh et al., [42] have observed that most CS journal papers tend to focus on specific sub areas of CS research, and therefore it is not surprising that there are few articles which focus on the discipline as a whole. *IEEE software* and *IEEE Transactions on Software Engineering* have very different editorial policies the former being a practice focused journal, the latter strongly research focused. Computer Science journals are less likely than educational or Information Systems journals to accept research with a critical perspective. Editors operate to restrict the discourse by imposing a particular style and philosophy, which excludes the voices of those outside the paradigm.
- Referees, reviewers, conference organising committee members to moderate what is the topic of discourse, and what may be said/published
    - Such groups control conference structure, themes, attendance/ invitations and nominate further reviewers
    - Determine keynote speakers, paper and poster presenters

4.1.3.3 Systems of regulation and control

This principle is concerned with control of "the production and manipulation of knowledge objects, that is, those elements of a socially constructed reality which are taken to be relevant…" [17]. Control is exerted through several different mechanisms:

- The refereeing and reviewing processes for publication. A whole arcana of procedures and techniques surrounds this area, with distinctions made between refereeing, formal reviewing and reviewing within the ACM for instance. In NZ, the NACCQ conference (http://www.naccq.ac.nz) has now moved to a double blind reviewing process, to enable authors to claim credit for their work as 'quality assured' for both NZ and Australian research performance systems.
- Hierarchy of journals. A process of ranking of journals is quite common to indicate the best publishing avenues and where the best quality research may be found, cf. the studies of computing and software engineering discipline publications [26, 42, 30]. But this practice is not without fishhooks. Such criteria omit niche journals for those who are specialists within the field; and there are difficulties in comparability between ranking surveys.
- Abstracts and citation indexes. There are a series of citation indexes and abstraction services, such as the International Sciences Citation Index in which academic journal articles are reported. This makes them available for a wide variety of searches. Bibliometric studies may count number of citations in such indices, as evidence of quality research output [30].
- Hierarchy of research outputs. Some Universities have explicit hierarchies and points systems for research outputs with targets for staff to meet which relate to promotion and tenure/merit decisions etc. A Computing Research and Education (CoRE) survey conducted in June of this year within the Australian and New Zealand University computing communities has attempted to develop a ranking for computing related conferences. This initiative has been taken in order to offset the potential damage caused by the incoming Australian Research Quality Framework (RQF), the model for which makes reference to the use of metrics in assessing the quality and impact of research. The cover letter by John Lloyd of ANU observes that "the use of bibliometrics is problematic for ICT disciplines where the main method of dissemination is through conference and not journal literature. Secondly it may be valuable to have robust indicators from conference data for appointments and promotions". The draft which I saw proposed four conference tiers: with tier 1 being best in its field and populated by top academics; tier 2 showing real engagement with the global research community, lowish acceptance rates and a strong program committee reviewing the work; tier 3 with a diligent program committee, yet not regarded as an especially significant event, and whose main function is the social cohesion of a community; tier 4 – all the rest. In the version of the ranking spreadsheet which I saw, no computing education conferences were in the top tier, ACM computing education conferences (SIGCSE, ITiCSE) and the Australasian equivalent (ACE) were however in the second tier; IEEE frontiers in Education was not mentioned, and nor were ICER, Koli Calling or the NACCQ in NZ, (which now like Koli is run in association with ACM). E-Learning conferences (ED-MEDIA, ASCILITE) tended to be rated in the third tier. This ranking system will in due course impact on where Australasian scholars direct their work as the funding will tend to follow the prestige.
- These regimes may validate and formalise activities deemed to be research, but they also operate to control what is legitimate by also defining what is not valid. For instance the author speaking with an external faculty member, was told of a case where he had published an

article in an internationally refereed transportation journal. But since his discipline was economics, and this was not an economics journal the article was not included in his tally for the year, as it was deemed to be outside the field.

- Rewards and sanctions at institutional level or by mechanisms such as research assessment exercises, for publication record or lack thereof. In NZ Middleton asserts that the PBRF is beginning to shape the behaviour of education academics, with an earlier trend towards more practice focused degree teaching now being reversed. So "the PBRF could encourage a downgrading of the grassroots engagements traditionally carried out by education [academics] with teachers and classrooms and prioritise for all [academics] publication in remote, overseas intellectual journals" [37]. As a by product, local publications and communities are also being devalued in favour of the global.
- Lack of reward for teaching performance. In some institutions effective teaching performance is merely taken as a given. Rather than an activity to be improved, valued and explicitly rewarded, teaching can become the poor cousin of research activity. The RAE in the UK is reported to have significantly raised the importance of research and "this increase has been at the expense of teaching" [5].
- Systems of accreditation, accreditation panels, and degree programme external monitors bring other forms of control and regulation of research activity. AUT University's business school is preparing itself for AACSB and Equis business school accreditations, in search of the global market and prestige that accompanies such accreditation schemes. However, the new breed of academics being imported into a relatively new University will come almost exclusively from traditional Universities, with their own rather separate cultures. The profile of a degree teacher under these schemes is defined in terms of traditional University sector mores, wherein discipline based research is highly valued, as opposed to the traditional discipline teaching or practice informed research of AUT University's 'legacy' staff. Thus such schemes inherently bring with them a colonising bias.
- The non-recognition of research associated with developing new courses. This activity is considered professional practice only, and the scholarship of teaching is defined outside the realm of research. The course may subsequently be written about in some context for publication, and this reflective secondary activity may be considered research, as opposed to the active practice itself. Can this distinction really be warranted? Or is it solely dictated by the need to categorise, rank and circumscribe the activities that constitute research.

As highlighted in section 2.2 above, the underlying bias behind the OECD definitions, which specifically exclude teaching from the research category, is an economic one. This discourse sits on top of the discourse about research itself, and decrees that teaching is not a process of generating new (and potentially economically valuable) knowledge, save at the postgraduate levels. Teaching is thus not construed as transformative, innovative or a contributor to economic "progress," but at the lower levels seems to be thought of as simply a process of knowledge transmittal, or traditional objectivist pedagogy. But what of other pedagogies, such as the constructivist or collaborativist [32], wherein the learner and teacher both engage in a process of inquiry to discover new forms of knowledge? Is it not possible for undergraduate learning and teaching to constitute research? Or is research simply the thing that its various definitions decree it to be, and our role simply to operate within the prescribed boundaries of the discourse?

5. IMPLICATIONS FOR CS ED RESEARCHERS

As can be seen from the above exposition, the life of a computing educator is constrained by sets of often conflicting forces and controls. Research and teaching are viewed in a dichotomous relationship, rather than in a broader model of scholarship. The underlying research drivers are economic, which tend to value discipline based research, in a model of academy-industry relationships which has been termed "academic capitalism" [cf. 4]. This has brought with it "in fields with close connections to the market, a new hierarchy of prestige and privilege…referenced to criteria external to the university…aspiring and rising academics in these fields remained however, subject to the academy's more traditional valuations of quality and prestige" [4]. Particular fields noted in [4] as having "close affinity to the market" include among others "computers and telecommunications".

In addition to such external valorizing of discipline based research, academic promotion and performance assessment systems also tend to value research above teaching.

5.1 Motivations for CS Education Research

5.1.1 Scholarship

Yet what is the reality of an academic's life? The proportion of time an average academic spends on teaching is typically equal to or greater than that spent on research. For instance [5] notes that Otago University in New Zealand (an established and traditional PhD granting research intensive institution), "has adopted a generic workload model for its Division of Humanities that recommends that 40 percent of an academic's time be spent on research and 40 percent on teaching (with the remaining 20 per cent being designated for service to university and community)."

Given this reality of academic life, the notion that our discipline teaching be research informed seems a logical corollary for any active scholar. We can add to this a simple professional duty of care, to teach our students as best we can, in a volatile and demanding subject.

5.1.2 Economic Return

However in a more utilitarian vein we can also argue the economic value of CS Ed research. The ACM offshoring study [6], notes that there were some 3.15 million people employed in IT occupations in the US in 2004. India graduates some 75,000 students annually from bachelor and masters degrees in computing and electronics, with a further 350,000 from other science and engineering fields at Universities and Polytechnics, many of whom enter the IT field upon graduation [6, p.35]. In 2001 China graduated 219,000 students in engineering and 120, 000 in science, and is now training about 100,000 per year for the software industry. In New Zealand with its limited population of some 4.1 million, the numbers are obviously smaller but still significant with 23,000 employed 'IT professionals' and 1800 IT degrees awarded in 2003 [19].

Therefore globally CS education may impact a million or more students per year. Work by Morrison and colleagues [38], applying the some econometric analyses used by the Australian

Government, has further suggested that the GDP return on higher level education of ICT students is some six times the net present value of the investment. It is therefore both a public and private concern that the quality of this education be sound.

Given the rapid rate of change in the computing disciplines, the number of still open questions and our continued challenges in teaching these disciplines well, CS Education Research has much to offer in this respect. If we wish to emphasise the economic argument, it could be said that this is a research field with the ability to contribute both to a multi-billion dollar industry (IT related higher education) and to transform economies by producing graduates capable of unleashing the innovative potential in the new systems, processes, products, technologies and industries to be gained from such investment.

5.2 The Need for High Quality CS Ed Research

5.2.1 The state of the art

One brief overview of typical CS Ed research can be found in [50] which concluded that articles presented at ACM SIGCSE technical symposium fell into 6 categories: *Experimental* – "where the author made any attempt at assessing the 'treatment' with scientific analysis"; *Marco Polo* – "I went there and I saw this"; *Philosophy* – where the author has made an attempt to generate debate of an issue on philosophical grounds; *Tools* - development of software or techniques for courses or topics; *Nifty* – a whimsical category with innovative, interesting ways to teach students our abstract concepts; *John Henry* – describing a course that seems "so outrageously difficult" as to be suspect, and charitably "at the upper limit of our pedagogy". The proportions of papers in each category over a twenty year period were found to demonstrate a relatively stable pattern, other than a noticeable shift from Marco Polo toward the tools category. Approximately 20% of CS Ed papers were in the so-called 'experimental' category. This would suggest that only 20% of papers in this major CS Ed conference can lay a claim to being regarded as CS Ed research. For some participants the technical symposium is essentially viewed as a 'swap-meet', so the high representation of purely descriptive 'Marco Polo' papers is a useful means for sharing ideas on how to teach a course in a rapidly evolving discipline. However it makes it difficult to stake a claim for the quality of the research being presented. Of more concern to me is the woeful lack of references to prior work in many papers of this type, and the constant repetition of local stories, which makes me wonder about the consistency of the reviewing process and how the papers met the criterion of novelty. Similar concerns have been noted in [43], noting the absence of literature review in many papers. In passing, they also expose a methodological flaw in the work of [50] itself, for failing to provide "estimates of reliability about his categorizations".

5.2.2 Towards Quality Research

Therefore, in our pedagogical research, if we wish to do quality work, we must also perform as well as we do in our discipline based research. As Shulman exhorts, "We don't judge each other's research on the basis of casual conversations in the hall; we say to our colleagues. 'that's a lovely idea! You really must write it up'. It may in fact take two years to write it up. But we accept this because it's clear that scholarship entails an artifact, a product, some form of community property that can be shared, discussed, critiqued, exchanged, built upon. So if pedagogy is to become an important part of scholarship, we have to provide it with this same kind of documentation and transformation" [48].

Yet as observed in [46] "methodologies generally used in computer science do not prepare us for the research questions that are relevant to CER (CS Ed Research)". Evaluation of educational innovations is far from straightforward, and we need to exercise care both in design of our CS Ed research projects and in analysis and evaluation of the outcomes. Good recommendations for evaluation can be found in the following sources [3, 7, 21, 27, 41].

Other strategies CS educators might adopt include: taking a suitable postgraduate research methods course in their own institution, to plug the gaps in knowledge; or volunteering to review for conferences, which is also a good way to become familiar with the CS Ed literature, to contribute to CS Ed community building, and to develop insight as a researcher.

One approach now being adopted by CS Ed researchers is the use of multi-institutional studies [cf. 22] to gather expertise, grow the scale of studies and build strength in analysing data to achieve more generalisable results than the typical single institution one-off studies that have been all too prevalent in CS Ed research. A further dimension of this work, through the Bootstrapping, BRACE and BRACELet projects has been a conscious CS Ed Researcher development programme, by associating novice and intermediate researchers, with more senior researchers in a supportive team environment. I would encourage newer researchers to join in such collaborative projects as they show much promise and offer a great learning environment.

There are other mechanisms by which the CS Ed Research community engages in creating "societies of discourse". In addition to CS Ed ACM conferences such as ITiCSE and SIGCSE, IEEE has the FIE conference and ICALT (an e-learning focused conference), a new CS Ed Research oriented conference has been launched (ICER), and regional conferences are evolving their international linkages (viz. ACE in Australasia, Koli in Finland, and NACCQ in New Zealand – each of which is now conducted in cooperation with ACM and SIGCSE). The ITiCSE conference runs a working group concept in which interested researchers may join with colleagues to write a report on topics of interest. This may offer an opportunity for mentoring of novice researchers and to gain exposure to the application of new research methodologies cf. [36] as one such example. An active phenomenographic group has also been holding PHICER workshops alongside the key conferences. Research groups in CS Ed Research exist, such as CSERGI which has linkages in several countries, and CETUSS which has been originated at Uppsala. Tapio Salakoski introducing last year's Koli Calling conference proceedings, also proclaimed the broader intention of founding a European association for computing education research.

6. CONCLUSION

This paper has highlighted the ways in which the "discourse of research" operates systematically through an innate bias against valuing educational research, to constrain the lives of CS Ed Researchers. Yet CS Education represents an important research field, with the ability to contribute significantly to the quality of education of the million or more students globally who study IT each year. Not only is this an important professional imperative for CS educators, but it has significant financial implications for all the stakeholders of global IT

education, and for the economies of the affected countries through the innovative potential of the resulting IT graduates.

Therefore, as a transdisciplinary research domain, CS Ed Research needs to write its own discourse. The work by Fincher & Petre [21], Pears et al., [41] and Seidman et al., [46] on discipline formation are good examples of CS Ed Research defining itself as a 'discipline', distinct from both the Education and CS disciplines, though cognate with CS and integral to quality CS education. Thus CS Ed Research can contribute to a vibrant teaching–research nexus in CS Education. In a recent study of educators beliefs about the relationship between research and teaching, one group of beliefs identified were that "Teaching and research share a symbiotic relationship in a learning community" [45]. This would be a magnificent mantra for all CS Educators to adopt. To teach the dynamic subject of CS well, educators now need to seriously add CS Ed Research to their research portfolios. It is time to move beyond descriptive conference papers giving merely anecdotal reports of experiences, and even transcend a reflective practice model of teaching. It is time to better inform CS Ed classroom practices by using appropriate research methods, based upon defensible models and research findings. If CS Ed Research is to be valued we need to rewrite the discourse, by positioning CS Ed Research as a research field noted for the rigour, quality and impact of its work, not simply a field in which over-worked CS educators may easily get a publication, without having to engage in the rigours of doing 'real' research.

We need to work together as colleagues to reduce isolation, share expertise, teaching materials, research techniques, and even data where appropriate. By developing a strong community of actively publishing researchers, who link quality CS Ed research with their quality teaching practice, the status of the CS Ed Research discipline will deservedly build, regardless of the opposition.

Yet let us acknowledge that status in itself is only a form of power arising from knowledge and engineered through discourse. This paper has demonstrated how the discourse relating to research is constructed and sustained. While entrenched, this discourse is open to manipulation, and as with all political apparatuses is not immune from being subverted. Some of the above mechanisms can be used to advantage in creating new disciplines, communities, events and publishing opportunities. Through deliberate action that consciously manipulates the levers of the discourse shaping CS Ed Research, we can shape the broader discourse surrounding research to our own ends.

7. ACKNOWLEDGMENTS

I wish to thank Anders Berglund for his generous support and the opportunity to present this paper at the Koli Calling conference. I also thank Professor Carmel McNaught and Alison Young for their insightful feedback on an earlier draft of this paper.

8. REFERENCES

[1] ABET. Criteria For Accrediting Computing Programs - Effective for Evaluations During the 2006-2007 Accreditation Cycle, Accreditation Board for Engineering and Technology, Inc. Computing Accreditation Commission, Baltimore, 2006

[2] Alcorn, N., Bishop, R., Cardno, C., Crooks, T., Fairbairn-Dunlop, P., Hattie, J., Jones, A., Kane, R., O'Brien, P. and Stevenson, J. Enhancing Education Research in New Zealand: Experiences and Recommendations from the PBRF Education Peer Review Panel. *New Zealand Journal of Educational Studies* (2), (2004).

[3] Almstrum, V., Dale, N., Berglund, A., Granger, M., Little, J.C., Miller, D., Petre, M., Schragger, P. and Springsteel, F., Evaluation: turning technology from toy to tool. Report of the working group on Evaluation. in *Integrating Technology into Computer Science Education Conference*, (Barcelona, Spain, 1996), ACM, 201-217.

[4] Anderson, M.S. The Complex Relations Between the Academy and Industry: Views from the Literature. *The Journal of Higher Education,72* (2), (2001), 226-247.

[5] Ashcroft, C. Performance Based Research Funding: A Mechanism to Allocate Funds or a Tool For Academic Promotion? *New Zealand Journal of Educational Studies,40* (1), (2005), 113-129.

[6] Asprey, W., Mayadas, F. and Vardi, M. Globalization and Offshoring of Software - A Report of the ACM Job Migration task Force, ACM, New York, 2006, 1-286.

[7] Bain, J. Introduction (to the special Issue on Evaluation). *Higher Education Research & Development,18* (2), (1999), 165-172.

[8] Banville, C. and Landry, M. Can the Field of MIS be disciplined? *Communications of the ACM,32* (1), (1989), 48-60.

[9] Barley, S., Meyer, G. and Gash, D. Cultures of Culture: Academics, Practitioners and the Pragmatics of Normative Control. *Administrative Science Quarterly,33,* (1988), 24-60

[10] Bishop, R. He Whakawhanaungatanga: The Rediscovery of a Family. in Bishop, R. ed. *Collaborative Research Stories, Whakawhanaungatanga,* Dunmore Press, Palmerston North, 1996, 35-71.

[11] Boyer, E. *Scholarship Reconsidered: Priorities Of The Professoriate - Carnegie Foundation Special Report.* Princeton University Press, Princeton, 1990.

[12] Carter, P. Building Purposeful Action: action methods and action research. *Educational Action Research,10* (2), (2002), 207

[13] Chang, C., Denning, P., Cross_II, J., Engel, G., Sloan, R., Carver, D., Eckhouse, R., King, W., Lau, F., Mengel, S., Srimani, P., Roberts, E., Shackelford, R., Austing, R., Cover, C.F., Davies, G., McGettrick, A., Schneider, G.M. and Wolz, U. Computing Curricula 2001 Computer Science, Joint Task Force IEEE-CS, ACM. New York, 2001, 1-201.

[14] Clark, M. Computer Science: a hard-applied discipline? *Teaching in Higher Education,8* (1), (2003), 71-87.

[15] Clear, T. Critical Enquiry in Computer Science Education. in Fincher, S. and Petre, M. eds. *Computer Science Education Research: The Field and The Endeavour,* Routledge Falmer, Taylor & Francis Group, London, 2004, 101- 125.

[16] Clear, T. TEAC Research Funding Proposals Considered Harmful: ICT Research at Risk. *NZ Journal of Applied Computing and IT,7,* (2003), 23-28.

[17] Davies, L. and Mitchell, G. The Dual Nature of the Impact of IT on Organizational Transformations. in Baskerville, R., Smithson, S., Ngwengyama, O. and DeGross, J. eds.

Transforming Organisations with Information Technology, Elsevier Science IFIP, North Holland, 1994.

[18] Denning, P. The Profession of IT - Crossing The Chasm. *Communications of the ACM,44* (4), (2001), 21-25.

[19] DOL. Information Technology Professional: Occupational Skill Shortage Assessment, Department of Labour, Wellington, 2005, 1-9. Retrieved 10/03/2006 from http://www.dol.govt.nz/PDFs/professional-report-it.pdf.

[20] Dougherty, J., Horton, T., Garcia, D. and Rodger, S. Panel on teaching faculty positions. *ACM SIGCSE Bulletin , Proceedings of the 35th SIGCSE technical symposium on Computer science education SIGCSE '04,36* (1), (2004), 231 – 232.

[21] Fincher, S. and Petre, M. *Computer Science Education Research: The Field and The Endeavour*. Routledge Falmer, Taylor & Francis Group, London, 2004.

[22] Fincher, S., R Lister, T Clear, Robins, A., Tenenberg, J. and Petre, M. Multi-Institutional, Multi-National Studies in CSEd Research: Some Design Considerations and Trade-offs. in Anderson, R., Fincher, S. and Guzdial, M. eds. *The First International Computing Education Research Workshop*, ACM, University of Washington, Seattle, WA, 2005, 111-121.

[23] Foucault, M. (ed.), *Power/Knowledge Selected Interviews and Other Writings 1972 -1977*. Pantheon, New York, 1980.

[24] Foucault, M. *The Archaeology of Knowledge and the Discourse on Language*. Pantheon, New York, 1972

[25] Gal-Ezer, J. and Harel, D. What (Else) Should CS Educators Know. *Communications of the ACM,41* (9), (1998), 77-84.

[26] Glass, R., Ramesh, V. and Vessey, I. An Analysis of Research in computing disciplines. *Communications of the ACM,47* (6), (2004), 89-94.

[27] Gunn, C. They Love it, but do they Learn From It? Evaluating the Educational Impact of Innovations. *Higher Education Research & Development,18* (2), (1999), 185 – 199.

[28] Habermas, J. *Knowledge and Human Interests, Theory and Practice, Communication and the Evolution of Society*. Heinemann, London, 1972.

[29] Heshusius, L. Freeing Ourselves from Objectivity: Managing Subjectivity or Turning Towards a Participatory Mode of Consciousness. *Educational Researcher* (Apr), (1994), 15-22.

[30] Katerattanakul, P., Han, B. and Hong, S. Objective Quality Ranking of Computing Journals. *Communications of the ACM,46* (10), (2003), 111-114.

[31] Knight, J. and Leveson, N. Should Software Engineers be Licensed? *Communications of the ACM,45* (11), (2002), 87-90.

[32] Leidner, D. and Jarvenpaa, S. The Use of Information Technology to Enhance Management School Education: A Theoretical View. *MIS Quarterly* (September), (1995), 265-291.

[33] Lévy-Leblond, J. Two Cultures or None? *The Pantaneto Forum* (8), (2002), 1-6. Retrieved 10/08/2006 from http://www.pantaneto.co.uk/issue2008/levyleblond.htm.

[34] Liegle, J. and Johnson, R. A Review of Premier Information Systems Journals for Pedagogical Orientation. *Information Systems Education Journal,1* (8), (2003), 1-10.

[35] Lister, R. Call Me Ishmael: Charles Dickens Meets Moby Book. *SIGCSE Bulletin*,38 (2), (2006), 11-13.

[36] Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Reilly, A.L., Sanders, K., Schulte, C. and Whalley, J. Research Perspectives on the Objects-Early Debate [Forthcoming]. *SIGCSE Bulletin,38* (4), (2006), tba.

[37] Middleton, S. Disciplining the Subject: The Impact of PBRF on Education Academics. *New Zealand Journal of Education Studies,40* (1), (2005)

[38] Morrison, I., Keynote Address - ICT, The Information Economy and Education. in *14th Annual Conference of the NACCQ*, (Napier, 2001), NACCQ.

[39] Mulder, F., van Weert, T.J. [eds] (2000) ICF-2000:Informatics Curriculum Framework 2000 for higher education. Paris, UNESCO / IFIP, 1-147. [URL:http://www.ifip.or.at/pdf/ICF2001.pdf]

[40] OECD. *The Measurement of Scientific and Technological Activities. Proposed Standard Practice for Surveys on Research and Experimental Development (Frascati Manual)*, OECD, Paris, 2002, 1-254.

[41] Pears, A., Seidman, S., Eney, C., Kinnunen, P. and Malmi, L. Constructing a core literature for computing education research. *SIGCSE Bulletin,37* (4), (2005), 152 – 161.

[42] Ramesh, V., Glass, R. and Vessey, I. Research in computer science:an empirical study. *The Journal of Systems & Software,70*, (2004), 165-176.

[43] Randolph, J., Bednarik, R. and Myller, N., A Methodological Review of the Articles Published in the Proceedings of Koli Calling 2001-2004. in *Koli Calling Conference on Computer Science Education*, (Koli, Finland, 2005), 103-108.

[44] Roberts, G. Review of Research Assessment: Report by Sir Gareth Roberts to the UK Funding Bodies, Joint funding bodies' Review of research assessment, London, 2003, 1-100. Retrieved 2/04/2006 from http://www.ra-review.ac.uk/reports/roberts/roberts_annexes.pdf.

[45] Robertson, J. and Bond, C. Experiences of the Relation between Teaching and Research: what do academics value? *Higher Education Research & Development,20* (1), (2001), 61-75.

[46] Seidman, S., Pears, A., Eney, C., Kinnunen, P. and Malmi, L., Maintaining a Core Literature of Computing Education research. in *Koli Calling Conference on Computer Science Education*, (Koli, Finland, 2005), 170-173.

[47] Shackelford, R., Cassel, L., Cross, J., Davies, G., Impagliazzo, J., Kamali, R., Lawson, E., LeBlanc, R., McGettrick, A., Slona, R., Topi, H. and vanVeen, M. Computing Curricula 2005 The Overview Report including The Guide to Undergraduate Degree Programs in Computing, Joint Task Force ACM, AIS, IEEE-CS, New York, 2005, 1-46

[48] Shulman, L. Teaching as community property; putting an end to pedagogical solitude. *Change,25* (6), (1993), 1-3.

[49] TEC. Performance-based Research Fund Guidelines 2006, Tertiary Education Commission, Wellington, 2005, 1-245

[50] Valentine, D., CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. in *SIGCSE Technical Symposium Proceedings (SIGCSE'04)*, (Norfolk, VA, 2004), ACM, 255-259.

[51] von_Tunzelmann, N. and Mbula, E. Changes In Research Assessment Practices In Other Countries Since 1999: Final Report, Joint funding bodies' Review of research assessment, London, 2003, 46. Retrieved 2/04/2006 from http://www.ra-review.ac.uk/reports/Prac/ChangingPractices.pdf.

[52] Zeni, J. Ethical Issues and Action Research. *Educational Action Research,6* (1), (1998), 9-19

Invited Seminar

The Development of Computer Science: A Sociocultural Perspective

Matti Tedre
University of Joensuu
Department of Computer Science and Statistics
P.O.Box 111, 80101 Joensuu, Finland
matti.tedre@cs.joensuu.fi

ABSTRACT
Computer science is a broad discipline, and computer scientists often disagree about the content, form, and practices of the discipline. The processes through which computer scientists create, maintain, and modify knowledge in computer science—processes which often are eclectic and anarchistic—are well researched, but knowledge of those processes is generally not considered to be a part of computer science. On the contrary, I argue that understanding of how computer science works is an important part of the knowledge of an educated computer scientist. In this paper I discuss some characteristics of computer science that are central to understanding how computer science works.

Keywords
Social studies of computer science; social issues; meta-knowledge in computer science

1. INTRODUCTION
Computer science is often taught in modules that dissect the discipline neatly into separate segments. Many of those segments seem quite unconnected—think about such ACM/IEEE computing curriculum's core topics as discrete structures, operating systems, human-computer interaction, and programming languages [8]. Although each core topic has a history of its own, if one takes a closer look at the history of computing it seems that each of the core topics in computing has gone through a number of stages before gaining a status of a core topic. In many cases those stages have included disdain and outright rejection, gradual maturing and academic approval, and finally a recognition as an important and integral part of computing field. In fact, every single core topic in the ACM/IEEE curriculum has been contested by someone, at some moment of time. For instance, pioneers of computing went to great lengths arguing that the theoretical parts of computer science are not a branch of mathematics but a central part of a new discipline called computer science [10,15]. The academic status of many technical things, such as programming, was still rejected in the ACM curriculum '68 [2].

Portraying a holistic view about the intellectual origins and about the development of branches of computing should offer computer science students a better understanding of the discipline than teaching the discipline as a collection of loosely connected islands. A disciplinary understanding requires understanding not only *what* computer science is and *how* its subjects are researched, but also *why* computer science is what it is and how did it get its current form. Learning that there have been methodological, epistemological, and other kinds of intellectual disagreements throughout the whole existence of computer science might give computer science students a perspective into the debates about computer science today. In this paper I outline a number of features in the development of computing as a discipline; features that may shed light on current debates, too: growth of technological momentum, the mangle of practice, and an anarchistic view of science. It does not matter whether one likes those characteristic of computer science or not, understanding those characteristics is an important part of understanding computer science [23].

2. TECHNOLOGICAL MOMENTUM
Historians of computing quite unanimously agree that the circumstances in which the stored-program paradigm was born were highly contingent. Those circumstances were brought together by the political situation, the war effort, advancements in science and engineering, new innovations in instrumentation, interdisciplinarity, influential individuals, coincidences, a disregard for costs, and a number of other sociocultural factors [5,7,20]. Many other development steps in modern computer science—such as the birth of high-level programming languages—have also been attributed to a number of influential sociocultural factors [3,4,22]. Computer science, as we know it, is a sociocultural construction that has been born, nurtured, and raised in a certain society. Today many concepts and developments in computer science, such as the stored-program paradigm and high-level programming languages, have become a part of the relatively stable core of computer science. At the same time, they have lost their human character and they are even sometimes perceived as something other than human constructs.

Thomas Hughes, who is a historian of science, has used the term *technological momentum* to refer to the tendency that young technological systems exhibit characteristics of social construction, but the more widespread and more established technological systems become, the more characteristics of technological determinism they exhibit [14]. In other words, the creation and recognition of newly-born innovations is a collective and subjective process, but usually the older and more prevalent innovations become, the more rigid, less responsive to outside influences, and thus more deterministic by nature those innovations become.

For instance, the stored-program paradigm (the constellation of innovations that surround the stored-program architecture) has gained enough technological momentum that it is today largely taken as an unquestioned foundation—even a sine qua non—of successful automatic computation (which might not be what the pioneers had in mind—for instance, the objective of Turing in his famous 1936 paper [25] was not to explain the limits of machine computation, but to specify the simplest machine that can perform any calculation that can be performed by a human

mathematician who has unlimited time, and who works with paper and pencil in accordance with some "rule-of-thumb" or rote method [6]).

The growth of technological momentum can be seen throughout the history of modern computing technology, too. In the early days of electronic digital computing, most computing machines of the time differed from each other in their architecture, design, constraints, and working principles. Over the course of time knowledge about the directions of computing accumulated, systems became more interdependent, and computing researchers and computer manufacturers increasingly followed traditional paths which others had tread. The first united architecture, the IBM System/360, marked a definite turning point in the change from social constructionism to technological determinism in computing machinery. Similar, the early construction of FORTRAN was directed by sociocultural and personal motivations, as well as economical and institutional considerations; but the more FORTRAN developed and the wider it spread, the more it institutionalized and the more rigid it became [21].

No matter how monumental some things in computing (such as the stored-program paradigm and high-level programming languages) seem today, there was a time when they were opposed by the scientific community. For instance, many authorities in the scientific establishment resisted ENIAC, which is now considered to be one of the most important milestones in the history of electronic computing [5,11,16,27]. After the construction of ENIAC and EDVAC there was still great uncertainty about the research directions and paradigms of electronic computing, and the first twenty years of electronic computing saw a great variety of fundamentally different competing technologies [27]. Similar, the first programming languages were shunned by the computing establishment but eventually some people, most notably Grace Hopper and John Backus, succeeded in selling the concept of programming languages to administrative, managerial, and technical people by arguing that programming languages would result in economic savings [3,4,22].

3. THE MANGLE OF PRACTICE

Although the growth of technological momentum in computing fields is quite visible in a number of examples, nothing has yet been said about the mechanisms of that growth. The growth of technological momentum in computer science does not work in any straightforwardly deterministic manner, and the growth of knowledge in computer science does not work according to any single philosophy of science. There are neither rigid rules that computer scientists would always stick to, nor a commonly held understanding of what computer science properly is. Rather, computer science is a logico-mathematical and technological enterprise driven by a diversity of needs, aims, agendas, and purposes; it is constructed and maintained in a complex mesh of institutions, social milieux, human practices, economical concerns, agenda, ideologies, cultures, politics, arts, and other technological, theoretical, and human aspects of the world.

Technological and intellectual changes in computer science can be reflected against Andrew Pickering's concept *mangle of practice* [18], which describes the development of science and technology as an ongoing cycle of development and revision of (1) theories and models, (2) the design and theory of instruments and how they work, and (3) the instruments themselves. When a computer scientist works, usually things do not go as planned—the world resists. He or she accommodates to this resistance by revamping some or all parts of the theoretical-technological structure, and tries again. In the end, the computer scientist hopes to get a robust fit between the theoretical and technological elements of a research study. In addition, researchers often need to accommodate for sociocultural factors, too—technoscience is not developed in a vacuum but within a dynamic network of societal, economic, cultural, institutional, ideological, political, philosophical, and ethical factors. That is, what we build and develop is in numerous ways guided by non-technical considerations and influences.

The birth of the stored-program paradigm is a prime example of the mangle in computing. The researchers who finally came up with the stored-program paradigm had their theories of computation, their prototype machines, and their theories of how their machines should work. They had to deal with numerous dead-ends; had to devote considerable effort to developing peripheral components, test equipment, and component technologies; had to revise their theories and concepts often; and had to spend a great deal of effort convincing other stakeholders about the value of computing [1,5,27]. Through numerous accommodations; such as revisions of theories, modifications of components, and rebuilding of instruments; the researchers at Moore School gradually arrived at the stored-program concept [5]. They certainly did not follow any pre-determined, scripted, or rigid approach but flexibly adapted to new situations, problems, and opposition; and changed their premises and approaches accordingly. Practical computer science combines receptivity and tenacity—it seems that practicing computer scientists can, at the same time, choose freely from a smörgåsbord of disciplines and approaches, yet be stubborn in the face of multidisciplinary opposition.

The mangle of practice in computer science is a continuous cycle of proposed theories, techniques, or mechanisms; corrections to theories, honing of techniques, improvements of mechanisms; negotiations between stakeholders, debates, power struggles; and other kinds of social and technical processes. In the course of time—and in the mangle—many theories are discredited and forgotten, many techniques become outdated, and many mechanisms turn out obsolete, but the knowledge that researchers in the computing science community gain through the mangle is gradually crystallized into a relatively stable core of computer science.

4. AN EFFICIENT ANARCHY

Methodological concerns have never played a leading role in computer scientists' work. Computer scientists publish relatively few papers with experimentally validated results [24], and research reports in computing rarely include an explanation of the research approach in the abstract, key word, or research report itself [26]. Computer science students usually learn their research skills from mentoring by their professors and from imitating previous research [23]. Typical computer science curricula, such as CC2001 [8], do not include courses on research methodology—yet in their work computer scientists do utilize a wide array of research methods, and often they combine methods in order to gain a wider perspective on the topic.

In a sense, many computer scientists might be characterized as *bricoleurs*—as researchers who work between competing paradigms. But from a disciplinary point of view, the diversity

of research approaches that are utilized in computer science renders computer science a methodologically and epistemologically eclectic discipline. Based on the methodological and epistemological nonconformity of computer scientists, computer scientists can perhaps be best characterized as opportunists. Their breaches of methodological norms and epistemological concerns are not an act of ignorance, though; there are calls for methodological regimentation in every single computer science forum (e.g., [13,17,24,26]). The best way to characterize the eclecticism and opportunism of computer science, as well as its conscious breaches of methodological and epistemological norms, is that computer science is an anarchistic enterprise.

Eclecticism, opportunism, and interdisciplinarity have not been detrimental to computer science, though. Eclecticism, opportunism, and interdisciplinary work were crucial in the shift from electromechanical computation to electronic computation; and interdisciplinarity also spurred new ideas within traditional disciplines [19]. An eclectic combination of incommensurable crafts and sciences creates an ontological, epistemological, and methodological anarchy, which inhibits detrimental dogmatism because no ontology, epistemology, or methodology can claim superiority over others. It may well be that superficial knowledge about powerful ideas actually enables researchers to utilize concepts or innovations without getting mired in field-specific debates. (Of course it also risks doing research in superficial, incorrect, and contradictory ways.)

Throughout the short history of computer science, computer scientists have quickly deepened the theoretical and technical knowledge about computing, and computer science has also worked as a catalyst in the creation of new research fields and spurred research in other disciplines. It seems that computer science has been efficient because of anarchism, not despite it. For example, the stored-program-paradigm was born as a result of a successful combination of a number of epistemologically and methodologically incompatible disciplines such as logic, electrical engineering, mathematics, physics, and radio technology. And especially after 1945 computer science has been influenced by a large and eclectic bunch of disciplines and approaches.

Anarchism can also be seen in that many innovations in computer science have been spurred despite the lack of support by the academic establishment, and sometimes even despite strong opposition by the establishment. In addition, without anarchism in computer science, the innofusion of many innovations in computer science could have been much slower, and some innovations might have not been introduced at all. For instance, in the 1970s computer scientists widely adopted Dijkstra's famous "GOTO statement considered harmful" hypothesis [9] without ever testing it empirically [12]. Whether computer scientists like it or not, anarchistic strands are thickly woven in computer science.

5. CONCLUSIONS

The processes of the creation, maintenance, and modification of computer science are complex and rich in nuances. Just teaching the topics of computer science and the basics of research in computer science does not yet create an educated academic computer scientist. Courses on the history of computing often focus on the milestones of computing, which might reinforce the idea of deterministic progress instead of portraying a living computer science. It is important to understand that many "milestone" concepts and events in computer science have in reality been far from discrete steps—the milestone concepts and events have often been multifaceted issues, and they have formed as a result of controversies, debates, and power struggles. The dynamics of computer science might be best revealed through social studies of computer science.

Sociohistorical understanding offers important "lessons learned"; it can be used to trace concepts and technologies to their origins in challenges, controversies, and discussions; and it enables one to discover parallels and analogies to modern technology that can be used for reassess current and future developments. Social studies of computer science can shed light on the very foundations of knowledge creation and maintenance in the computing disciplines—for instance, expose how the philosophical, theoretical, conceptual, and methodological frameworks of computer science are created, maintained, and managed. Social studies can explain how computer scientists and other stakeholders create computer science and computing technology. Social studies of computer science can reveal the human origins and human character of our science. Meta-knowledge is an inherent and important part of any academic discipline, including computer science.

6. ACKNOWLEDGMENTS

I wish to thank professor Erkki Sutinen, Dr. Esko Kähkönen, and professor Piet Kommers for their support and numerous comments on my work.

7. REFERENCES

[1] Aspray, W. Was Early Entry a Competitive Advantage? US Universities That Entered Computing in the 1940s. *IEEE Annals of the History of Computing* 22(3) (2000), 42-87.

[2] Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T- E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and Young, D. M. Curriculum '68, Recommendations for Academic Programs in Computer Science. *Communications of the ACM*, 11(3) (March 1968), 151-197.

[3] Backus, J. The History of FORTRAN I, II, and III. In *History of Programming Languages* (ed. Wexelblat, R. L.). Academic Press: London, UK, 1981, 25-45.

[4] Bright, H. S. Early FORTRAN User Experience. *IEEE Annals of the History of Computing* 6(1) (1984), 28-30.

[5] Campbell-Kelly, M., Aspray, W. *Computer: A History of the Information Machine (2nd ed.)*. Westview Press: Oxford, UK, 2004.

[6] Copeland, B. J., Proudfoot, D. What Turing Did after He Invented the Universal Turing Machine. *Journal of Logic, Language, and Information* 9(4) (2000), 491-509.

[7] Croarken, M. G. The Emergence of Computing Science Research and Teaching at Cambridge, 1936-1949. *IEEE Annals of the History of Computing* 14(4) (1992), 10-15.

[8] Denning, P. J., Chang, C. (chairmen) and the Joint Task Force on Computing Curricula. Computing Curricula 2001. *ACM Journal of Educational Resources in Computing*, 1(3) (Fall 2001), Article #1.

[9] Dijkstra, E. W. Go To Statement Considered Harmful. *Communications of the ACM* 11(3) (1968), 147-148.

[10] Dijkstra, E. W. Programming as a Discipline of Mathematical Nature. *American Mathematical Monthly* 81(June-July 1974), 608-612.

[11] Flamm, K. *Creating the Computer: Government, Industry, and High Technology*. Brookings Institution: Washington, D.C., USA, 1988.

[12] Glass, R. L. "Silver bullet" Milestones in Software History. *Communications of the ACM* 48(8) (2005), 15-18.

[13] Glass, R. L., Ramesh, V., Vessey, I., An Analysis of Research in Computing Disciplines. *Communications of the ACM* 47(6) (2004), 89-94.

[14] Hughes, T. P. Technological Momentum. In *Does Technology Drive History? The Dilemma of Technological Determinism* (eds. Smith, M. R. & Marx, L.). The MIT Press: Cambridge, Mass., USA, 1994, 101-114.

[15] Knuth, D. E. Computer Science and its Relation to Mathematics. *American Mathematical Monthly* 81(April 1974), 323-343.

[16] Marcus, M. and Akera, A. Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture. *IEEE Annals of the History of Computing* 18(1) (1996), 17-24.

[17] Palvia, P., Mao, E., Salam, A.F., Soliman, K. Management Information Systems Research: What's There in a Methodology? *Communications of the AIS* 11(16) (2003), 1-33 (Article 16).

[18] Pickering, A. *The Mangle of Practice: Time, Agency, and Science*. The University of Chicago Press: Chicago, USA, 1995.

[19] Puchta, S. On the Role of Mathematics and Mathematical Knowledge in the Invention of Vannevar Bush's Early Analog Computers. *IEEE Annals in the History of Computing* 18(4) (1996), 49-59.

[20] Pugh, E. W., Aspray, W. Creating the Computer Industry. *IEEE Annals of the History of Computing* 18(2) (1996), 7-17.

[21] Rosenblatt, B. The Successors to FORTRAN: Why Does FORTRAN Survive? *Annals of the History of Computing* 6(1) (1984), 39-40.

[22] Sammet, J. E. *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc.: Englewood Cliffs, New Jersey, 1969.

[23] Tedre, M. *The Development of Computer Science: A Sociocultural Perspective.* Ph.D. Thesis, University of Joensuu, Finland, 2006.

[24] Tichy, W. F., Lukowicz, P., Prechelt, L., Heinz, E. A. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software* 28(1995), 9-18.

[25] Turing, A. M. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2* 42(1936), 230-265.

[26] Vessey, I., Ramesh, V., Glass, R. L. Research in Information Systems: An Empirical Study of Diversity in the Discipline and Its Journals. *Journal of Management Information Systems* 19(2) (2002), 129-174.

[27] Williams, M. R. *A History of Computing Technology*. Prentice-Hall: New Jersey, USA, 1985.

Research Papers

Progress Reports and Novices' Understanding of Program Code

Linda Mannila
Dept. of Information Technologies, Åbo Akademi University,
Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linda.mannila@abo.fi

ABSTRACT

This paper introduces *progress reports* and describes how these can be used as a tool to evaluate student learning and understanding during programming courses. A progress report includes a short piece of program code (on paper), covering topics recently introduced in the course, and four questions. The two first questions are of a "trace and explain" type, asking the students to describe the functionality of the program using their own words - both line by line and as a whole. The two other questions aim at gaining insight into the students' own opinions of their learning, as they are asked to write down what they think they have learned so far in the course and what they have experienced as most difficult.

Results from using progress reports in an introductory programming course at secondary level are presented. The responses to the "trace and explain" questions were categorized based on the level of overall understanding manifested. We also analyzed students' understanding of individual programming concepts, both based on their code explanations and on their own opinions on what they had experienced as difficult. Our initial experience from using the progress reports is positive, as we feel that they provide valuable information during the course, which most likely would remain uncovered otherwise.

1. INTRODUCTION

In [5], we reported on a study from teaching introductory programming at high school level.[1] The results showed that abstract topics such as algorithms, subroutines, exception handling and documentation were considered most difficult, whereas variables and control structures were found rather straight forward. These results were well in line with those of other researchers (e.g. [6, 7]). In addition, our findings indicated that most novices found it difficult to point out their weaknesses. Moreover, exam questions asking the students to read and trace code showed a serious lack in program comprehension skills among the students. One year later (2005/2006) we conducted a new study, in which we further investigated these issues using what we have called *progress reports*. This paper presents the results from this study.

We begin the paper with a background section, followed by a section describing the study and the methods used. Next, we present and discuss the results, after which we conclude the paper with some final words and ideas for future work.

2. BACKGROUND

Introductory programming courses tend to have a strong focus on construction, with the overall goal being students getting down to writing programs using some high-level language as quickly as possible. This is understandable, since the aim of those courses is to learn programming, which is commonly translated into the competence of using language constructs. Being able to write programs is, however, only one part of programming skills; the ability to read and understand code is also essential, particularly since much of a programmer's time is spent on maintaining code written by somebody else. In this paper, we refer to reading a program as "the process of taking source code and, in some way, coming to 'understand' it" (p. 1), as defined by Deimel [3].

One could assume that students learning to write programs automatically also learn how to read and trace code. Research has, however, indicated that this is not always the case. For instance, Winslow [15] notes that "[s]tudies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies" (p. 21). In 2004, a working group at the ITiCSE conference [8] tested students from seven countries on their ability to read, trace and understand short pieces of code. The results showed that many students were weak at these tasks.

Why then is it difficult to read code? Spinellis [13] makes the following analogy: "when we write code, we can follow many different implementation paths to arrive at our program's specification, gradually narrowing our alternatives, thereby narrowing our choices [...] On the other hand, when we read code, each different way that we interpret a statement or structure opens many new interpretations for the rest of the code, yet only one path along this tree is the correct interpretation" (p. 85-86).

Fix et. al [4] cite Letovsky, according to whom the overall goals of a program often can be inferred when reading the code, based on for instance variable names, comments and other documentation. Letovsky also suggests that the same thing applies to the data structures and actions of the program (i.e. the implementation): a person reading a program understands the actions of each line of code separately. The difficulty arises when trying to map high-level goals with their representation in the code. Lister et al. [9] talk about students failing to "see the forest for the trees" (p. 119). They have found that weak students, in particular, seem to have difficulties in abstracting the overall workings of a program from the code. This finding is supported by Pennington [10], who has found that whereas experts infer what a program does from the code, less experienced programmers make speculative guesses based on superficial information such as variable names, without confirming their guesses in any way.

Pennington [11] has also developed a model describing program comprehension, according to which a programmer constructs two mental models when reading programming code. First, the programmer develops a *program model*, which is a low-level ab-

[1] In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16-19. The main objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies.

straction based on control flow relationships (for instance loops or conditionals). This program-level representation is formed at an early stage and is inferred from the structure of the program text. After that, the programmer develops a *domain model*, which is a higher-level abstraction based on data flow, containing main functionality and the information needed to understand what the program truly does. Similarly, Corritore and Wiedenbeck [2] have found that novices tend to have concrete mental representations of programming code (operations and control flow), with only little domain-level knowledge (function and data flow).

In 1982, Biggs and Collis [1] introduced the SOLO (Structure of the Observed Learning Outcomes) taxonomy, which can be used to set learning objectives for particular stages of learning or to report on the learning outcomes. The taxonomy is a general theory which was not originally designed to be used in a programming context. Lister et al. [9] have used the taxonomy to describe how code is understood by novice programmers, and describe the five SOLO levels applied to novice programming as follows:

Prestructural ”In terms of reading and understanding a small piece of code, a student who gives a prestructural response is manifesting either a significant misconception of programming, or is using a preconception that is irrelevant to programming. For example, [...] a student who confused a position in an array and the contents of that position (i.e. a misconception)” (p. 119).

Unistructural ”[...] the student manifests a correct grasp of some but not all aspects of the problem. When a student has a partial understanding, the student makes [...] an ’educated guess”’ (p. 119).

Multistructural ”[...] the student manifests an understanding of all parts of the problem, but does not manifest an awareness of the relationships between these parts - the student fails to see the forest for the trees” (p. 119). For example, a student may hand execute code and arrive at a final value for a given variable, still not understanding what the code does.

Relational ”[...] the student integrates the parts of the problem into a coherent structure, and uses that structure to solve the task - the student sees the forest” (p. 119). For instance, after thoroughly examining the code, a student may infer what it does - with no need for hand execution. Lister et al. also note that many of the relational responses start out as multistructural, with the student hand tracing the code for a while, then understanding the idea, and writing down the answer without finishing the trace.

Extended Abstract At the highest SOLO level, ”the student response goes beyond the immediate problem to be solved, and links the problem to a broader context” (p. 120). For example, the student might comment on possible restrictions or prerequisites, which must be fulfilled for the code to work orderly.

Lister et al. found that a majority of students describe program code line by line, i.e. in a multistructural way. They argue that students who are not able to read and describe programming code relationally do not possess the skills needed to produce similar code on their own.

3. THE STUDY

3.1 Data

The data analyzed in this study were collected when two high school student groups (25 students) took an introductory programming course in 2005/2006. The majority of the students had no programming background. The courses were taught using Python and covered the basics of imperative programming. The data collection was conducted using surveys, progress reports and a final exam; in this paper we focus the analysis on the progress reports and on parts of the post course survey.

3.1.1 Progress Reports

A progress report can be seen as a type of learning diary (on paper), aiming at revealing the students’ own opinions and thoughts about their learning. What differentiates it from a traditional diary is that in addition to calling for self reflection, the report makes it possible for the teacher to evaluate students’ understanding based on their responses to ”trace and explain” questions. In this study each report included a piece of code, dealing with topics recently covered in the course, and four questions. First, in the ”trace and explain” questions, the students were to read the code and in their own words (1) describe what each line of the code does, and (2) explain what the program as a whole does on a given set of input data. In addition, students were asked to write down what they had learned and what had been most difficult so far in the course.

The first progress report was handed out after 1/3 and the second after 2/3 of the course. In total, we have analyzed 50 progress reports (two for each of the 25 students) and 25 post course surveys.

3.2 Method

The progress reports and post course surveys were analyzed manually in order to study three questions: how do students (1) understand program code as a whole, (2) understand individual constructs, and (3) perceive the difficulty level of different programming topics.

The first two questions were addressed by grouping the explanations given for the ”trace and explain” questions according to qualitative differences found in the data. On this point, the study resembles the SOLO study by Lister et. al [14] to some extent. However, in this study, we have explicitly asked students for both a multistructural and a relational response for each piece of code, giving us the possibility to compare how well the two responses match for individual students.

Finally, to address the third question, we studied the difficulty level of topics as perceived by the students by looking at both the progress reports and the post course survey. The ”trace and explain” questions also provided data pertinent to this part of the study as explanation errors were considered indications of student experiencing problems with that specific topic.

4. RESULTS AND DISCUSSION

4.1 Program Understanding

The ”trace and explain” code given in the first progress report is listed below (algorithm 1).

Students were asked to explain what this piece of code does if two integers are given as input. The analysis of the overall explanations gave rise to four categories:

Algorithm 1 Program given in progress report 1

```
try:
    a = input("Input number: ")
    b = input("Input another number: ")
    a = b

    if a == b:
        print "The if-part is executed..."

    else:
        print "The else-part is executed..."

except:
    print "You did not input a number!"
```

1. Correct explanation (n = 13)
2. Choosing the wrong branch in the selection after not explaining the meaning of the statement `a = b` (n = 5)
3. Choosing the wrong branch although having explicitly explained the statement `a = b` (n = 3)
4. Giving an output totally different from the ones possible based on the code (n = 4)

The first category is straight forward: over half of the students gave a perfect overall explanation for the program code, not only listing the output but also explaining the reasons for why that specific output was produced. The second category covers responses in which the student had failed to explain what the `a = b` statement means, and hence also missed the fact that when arriving at the selection statement, `a` does, in fact, equal `b`.

More concerning is the third category, in which students who have explicitly explained the `a = b` statement still believe that the else-branch will be executed. This indicates a misunderstanding related to either the results of an assignment statement, or the workings of the selection statement.

The fourth category appears to be some kind of guessing; the student thinks that the program outputs something that might have been expected from the code (e.g. values instead of one of the text messages), but was nevertheless incorrect.

Algorithm 2 Program given in progress report 2

```
def divisible(x):
    if x % 2 == 0:
        return True
    else:
        return False

default = 5
number = default

while number > 0:
    try:
        number = input ("Give me a number: ")
        result = divisible(number)
        print result
    except:
        print "Numbers only, please!"
```

The piece of code included in the second progress report is shown in algorithm 2. For this program, students were given a list of input data including positive integers and a character, ending with a negative integer. The explanations were analyzed in a similar manner to the corresponding task in the first progress report, and four categories were found:

1. Correct explanation (n=8)
2. Not understanding what it means to return a Boolean value (n=10)
3. Missing the last iteration, otherwise correct (n=4)
4. Incorrect output (n=3)

The number of correct overall explanations for the program in the second progress report was smaller than for the first report. We had expected that subroutines would be perceived as difficult, since these are commonly one of the main stumbling blocks in introductory programming [5, 6, 7]. However, the analysis revealed that subroutines per se were not necessarily the main problem; instead surprisingly many students had difficulties in understanding what happens when a subroutine returns a Boolean value. The code in algorithm 2 checks if the input is divisible by two and outputs either `True` or `False` based on the result as long as the input number is positive. Common misunderstandings were for instance that returning `True` means that control is returned to the main program, whereas returning `False` makes the subroutine start all over again. Some students thought that there is no output whenever the subroutine returns `False`.

The third category indicate that some students had difficulties deciding when a while loop stops, missing the last iteration. It should be noted that the loop will be executed once more after a negative value is input.

The fourth category is similar to the one for the first progress report. That is, students seem to be guessing, stating that the program outputs something totally different (in this case the input values) from what the subroutine returns.

The categories found were quite similar for both progress reports, and can be related to the SOLO taxonomy as presented by Lister et al. [9]. The first category (correct explanation) contains relational responses, whereas the second and third ones (indicating a misunderstanding) can be seen as containing explanations at the pre- or unistructural level. The fourth category (guessing) includes unistructural responses, which can also be seen as the ”speculative guesses” mentioned by Pennington [10].

4.2 Understanding of Individual Statements

In order to further analyze students' skills to read and understand code, we analyzed how they explained individual statements related to a set of programming topics. The explanations found were of the following types:

- *Correct* - the student explained the statement ”by the book”
- *Missing* - the student did not write any explanation for that given statement
- *Incomplete* - the student's explanation was correct to some extent, but still lacking some parts
- *Erroneous* - the student gave an explanation that was ”not even close” (for instance indicating a misconception)

Although the number of students giving correct overall explanations for the programs decreased from the first to the second report (as shown in section 4.1), the results presented in the diagram in Figure 1 indicate that at the same time the students became better at understanding individual statements: the number of correct explanations for individual statements increased while the number of incomplete or missing explanations decreased. This can, however, be seen as quite natural as one could - and should - expect students to gain a better understanding for individual topics as the course goes on and they become more experienced and familiar with the topics. We found no erroneous comments related to individual topics included in both reports.

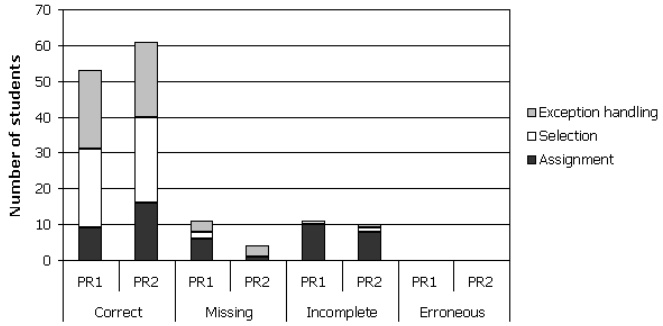

Figure 1: *The frequency of different types of explanations given by students for individual statements in the two progress reports.*

The second progress report introduced some new topics not present in the first one. The distribution of explanation types for these is illustrated in Figure 2. The data in the diagram reflect the previously mentioned difficulties related to subroutines returning Boolean values: almost half of the students gave incorrect explanations on this point. Interestingly, the other half of the students explained the returns correctly. This might imply that this topic (subroutine returns, at least for Boolean values) is something students either "get or do not get".

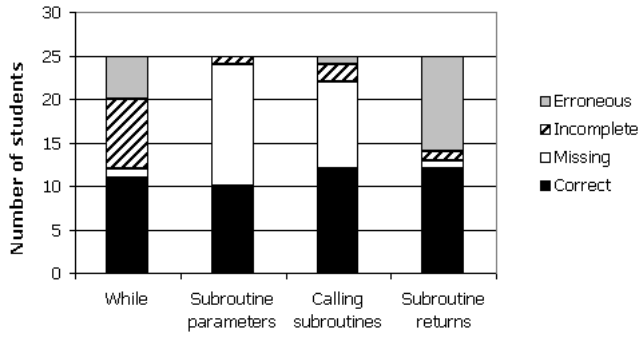

Figure 2: *The frequency of different types of explanations given by students for new topics in the second progress report.*

Many students did not explain subroutine calls or parameters, which makes it difficult to say anything about the perceived difficulty level of those topics. If all missing explanations indicate "erroneous explanations", the number of students not understanding subroutine calls and parameters is alarmingly high. On the other hand, if the missing explanations were due to students finding those aspects "self evident", and therefore not needing any explanation, the number of correct explanations for those topics would be high. When taking into account that the overall explanations to the code in the progress reports did not indicate any specific difficulties in calling subroutines with parameters, the latter explanation might be a bit closer at hand. These are, however, only speculations, and in our opinion the question of which aspects of subroutines make them difficult merits further investigation.

Having analyzed the explanations for both individual statements and entire programs, we can conclude that more students were able to correctly explain the program line by line than as a whole. This was found for both progress reports. When related to the levels in the SOLO taxonomy, most students were able to give correct explanations in multistructural terms, but only part of them did so relationally.

4.3 Difficulty of Topics

Apparently, assignment statements constituted the most common difficulty for students in the first progress report. However, this was not reflected in the students' own opinions on what they found difficult in the course at that time. Instead, they mentioned topics such as loops (n=4), the selection statement (n=2) and lists (n=3). Moreover, 40% of the students only gave an incomplete explanation for what `a = b` means, not mentioning values or variables, but stating for instance that "a becomes b" or "a is b". Clearly, such a student has some idea of what happens, but without mentioning values, this explanation is not exact enough.

In the second progress report, the problems students faced in the "trace and explain" questions (returns and subroutines) were in line with the difficulties they reflected upon in the other questions: almost half of the students stated that subroutines were most difficult. Some students still reported having problems with lists (n=2) and loops (n=2).

In the post course survey, students were asked to rate each course topic on the scale 1-5 (1 = very easy, 5 = very difficult). The results showed that the perceived difficulty levels were quite consistent with the corresponding results presented in our previous study [5]. Subroutines, modules, files and documentation were still regarded as most problematic (average difficulty of 2.8 - 3.2). There was, however, one exception: in the previous study, exception handling was also experienced as one of the most difficult topics (average of 2.9), but in the current study this was no longer the case (average of 2.1). The progress reports supported this finding: exception handling was not mentioned as a difficult topic at all, and nearly all students gave a perfect explanation for statements dealing with exception handling in the "trace and explain" questions. We were pleased to see this result, as we had made changes to the syllabus in order to facilitate students' learning of this particular topic. Exception handling was now one of the topics introduced at the very beginning of the course, together with variables, output and user input, and students got used to check for and deal with errors from the start. It thus seems as if the order in which topics are introduced does have an impact on the perceived difficulty level, as suggested by Petre et al. [12], who have found indications of topics being introduced early in a course to be perceived as "easy" by students, whereas later topics usually are considered more difficult.

5. CONCLUSION

In this paper we have introduced progress reports, and described how we have used these to analyze student understanding and progress during an introductory programming at high school level. Our initial experiences from using the reports are positive, as we feel that they provide important information during the course, which most likely would remain uncovered otherwise. The re-

ports can be used in various ways, and can be seen as a rather small active effort, with which one can collect valuable information that, if used wisely, can make a large difference for students learning to program.

Asking the students to fill out reports repeatedly throughout a course could, for instance, not only serve as a tool for continuous checkups of student progress throughout a course, but also as a starting point for individual discussions, in which the tutor/teacher and the student could go through the explanations and any potential errors. Naturally, such discussions would require extra resources in the form of teacher/tutor effort and time, which might not be available. A less demanding alternative would be for the teacher/tutor to only have short discussions with students that are evidently in need of help based on the report. At the same time they could try to find out where the difficulties truly lie - whether it is in the topics the student has written down, or somewhere completely different. Based on our findings, the latter would be most common, as the errors that actually occurred in students' code explanations did not always match the topics that were most problematic according to the students themselves. The difference was particularly evident in the first progress report, where most errors were related to assignment statements, but none of the students mentioned these as being difficult.

Some students seemed to be guessing when answering the "trace and explain" questions. In the future, we will add another question to the progress reports that ask the students to evaluate (e.g. on a given scale) how confident they are about their explanations. This will make it easier to distinguish between students truly believing in their answers and those merely guessing.

The results from the SOLO study presented by Lister et al. [14] are interesting, as they divide student responses into different SOLO categories. However, asking the students for both a multistructural and a relational response makes the data even more interesting, since it gives us two different responses for each program. These can be used to analyze how well the responses match for an individual student. As seen in the previous section, students were in general able to give perfect descriptions of the programs line by line, but only a fraction of these gave a perfect explanation of what the program did as a whole. This finding suggests that novice programmers tend to understand concepts in isolation, and is thus consistent with the results presented by Lister et al. [14] and with Pennington's idea of program vs. domain models [11].

As educators, we expect students to go through and learn from examples when we introduce a new topic. Doing so, the student's attention is on the construct, not on understanding how the given piece of code solves a particular problem. This means that we mainly support the development of a program model of understanding. For students to develop a more complete understanding of a program, we should also give them tasks and examples that facilitate them in the process of developing a solid domain model. The progress reports can be used as a feedback tool to help us evaluate how we are doing on this point.

The good results from introducing exception handling, a topic which was previously perceived as difficult, earlier in the syllabus were encouraging, and indicated that the order in which topics are introduced can make a difference. Since subroutines continue to be a problematic topic in introductory programming, we suggest that one would try to teach modular thinking and writing own, simple subroutines as one of the first topics in introductory programming courses.

6. ACKNOWLEDGEMENTS

Special thanks to Mia Peltomäki and Ville Lukka for collecting the data.

7. REFERENCES

[1] J. Biggs and K. Collis. *Evaluating the Quality of Learning - the SOLO Taxonomy*. New York: Academic Press, 1982.

[2] C. Corritore and S. Wiedenbeck. What Do Novices Learn During Program Comprehension. *International Journal of Human-Computer Interaction*, 3(2):199–208, 1991.

[3] L. E. Deimel and J. F. Naveda. Reading Computer Programs: Instructor's Guide and Exercises, 1990. Education materials. Available online: http://www.literateprogramming.com/em3.pdf. Retrieved August 29, 2006.

[4] V. Fix, S. Wiedenbeck, and J. Scholtz. Mental representations of programs by novices and experts. In *INTERCHI '93: Proceedings of the INTERCHI '93 conference on Human factors in computing systems*, pages 74–79, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press.

[5] L. Grandell, M. Peltomaki, R.-J. Back, and T. Salakoski. Why Complicate Things? Introducing Programming in High School Using Python. In D. Tolhurst and S. Mann, editors, *Eighth Australasian Computing Education Conference (ACE2006)*, CRPIT, Hobart, Australia.

[6] A. Haataja, J. Suhonen, E. Sutinen, and S. Torvinen. High School Students Learning Computer Science over the Web. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2001. Available online: http://imej.wfu.edu/articles/2001/2/04/index.asp. Retrieved August 29, 2006.

[7] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A Study of the Diffculties of Novice Programmers. In *ITICSE '05: Proceedings of the 10th annual ITiCSE conference*, pages 14–18, Capacrica, Portugal, 2005. ACM Press.

[8] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, 2004.

[9] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, 38(3):118–122, 2006.

[10] N. Pennington. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, pages 100–113, 1987.

[11] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

[12] M. Petre, S. Fincher, J. Tenenberg, et al. "My Criterion is: Is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. Technical Report 6-03, Computing Laboratory, University of Kent, UK, June 2003.

[13] D. Spinellis. Reading, Writing, and Code. *ACM Queue*, 1(7):84–89, October 2003.

[14] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. A. Kumar, and C. Prasard. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. In D. Tolhurst and S. Mann, editors, *Eighth Australasian Computing Education Conference (ACE2006)*.

[15] L. E. Winslow. Programming pedagogy, a psychological overview. *SIGCSE Bull.*, 28(3):17–22, 1996.

An Objective Comparison of Languages for Teaching Introductory Programming

Linda Mannila
Turku Centre for Computer Science
Åbo Akademi University
Dept. of Information Technologies
Joukahaisenkatu 3-5, 20520 Turku, Finland
linda.mannila@abo.fi

Michael de Raadt
Department of Mathematics and Computing and
Centre for Research in Transformational Pedagogies
University of Southern Queensland, Toowoomba
Queensland, 4350, Australia
deraadt@usq.edu.au

ABSTRACT
The question of which language to use in introductory programming has been cause for protracted debate, often based on emotive opinions. Several studies on the benefits of individual languages or comparisons between two languages have been conducted, but there is still a lack of objective data used to inform these comparisons. This paper presents a list of criteria based on design decisions used by prominent teaching-language creators. The criteria, once justified, are then used to compare eleven languages which are currently used in introductory programming courses. Recommendations are made on how these criteria can be used or adapted for different situations.

Keywords
Programming languages, industry, teaching

1. INTRODUCTION
A census of introductory programming courses within Australia and New Zealand [5] revealed reasons why instructors chose their current teaching language (shown in Table 1). The most prominent reason was industry relevance, before even pedagogical considerations. This suggests academics perceive pressure to choose a language that may be marketable to students, even if students themselves may not be aware of what is required in industry.

The primary objective of introductory programming instruction must be to nurture novice programmers who can apply programming concepts equally well in any language. Yet many papers from literature argue that one language is superior for this task. Such research asserts a particular language is superior to another because, in isolation, it possesses desirable features [2, 3, 4, 9, 21] or because changing to the new language seemed to encourage better results from students [1, 11]. What is shown in literature is surely only a reflection of the innumerable debates that have undoubtedly taken place within teaching institutions.

While the authors of this paper do not believe that language choice is as critical as choice of course curriculum used to deliver teaching, it is important to choose a language that will best support an introductory programming curriculum.

Table 1: Reasons for instructors' language choice

Reason	Count
Industry relevance/Marketable/Student demand	33
Pedagogical benefits of language	19
Structure of degree/Department politics	16
OOP language wanted	15
GUI interface	6
Availability/Cost to students	5
Easy to find appropriate texts	2

1.1 Background
The choice of programming language to use in education has been a topical issue for some time. In the early 1980s, Tharp [22] made a language comparison of COBOL, FORTRAN, Pascal, PL-I, and Snobol, primarily focused on efficiency of compilation and speed of code implementation, in order to provide educators with information needed to choose a suitable language. Today, considerations focus more on pedagogical concerns and the range of languages is even broader.

George Milbrandt suggests the following list of language features for languages used in high schools in [20].

- easy to use
- structured in design
- powerful in computing capacity
- simple syntax
- variable declaration
- easy input/output and output formatting
- meaningful keyword names
- allowing expressive variable names
- provide a one-entry/one-exit structure
- immediate feedback
- good diagnostic tools for testing and debugging

Many of the criteria in the list above are echoed by McIver and Conway [15] who list seven ways in which introductory programming languages make teaching of introductory programming difficult. They also put forward seven principles of programming language design aiming to assist in developing good pedagogical languages. Neither of these studies demonstrates application of these criteria to make comparison between languages.

Instruments to facilitate the process of choosing a suitable language have also been suggested (e.g. [18]), but without presenting any comparable results.

1.2 Goal
This paper is intended to be an objective comparison of common languages, based on design decisions used by prominent teaching-language creators, drawing conclusions that allow instructors to make informed decisions for their students. It is also intended to provide ammunition for those who are, for pedagogical reasons, seeking to make a language change, in an environment where industry relevance can be overvalued.

The following section lists the criteria used to make a comparison of languages in section 3. Finally conclusions are drawn in section 4.

2. CRITERIA

A list of seventeen criteria has been created and is presented in the following subsections. Each criterion has been suggested by creators of languages that are considered "teaching languages".

1. Seymour Papert (creator of LOGO) [1]
2. Niklaus Wirth (creator of Pascal) [2]
3. Guido van Rossum (creator of Python) [3]
4. Bertrand Meyer (creator of Eiffel) [4]

Each criterion is drawn from the design decisions made by each of these language creators as they describe their languages.

The criteria refer to languages in general. There is no mention of paradigm within the criteria and this allows comparison of languages across paradigms.

Criteria are grouped into related subsections for ease of application. The criteria are shown in no particular order of priority.

2.1 Learning

The following criteria relate the programming language to aspects of learning programming.

2.1.1 The language is suitable for teaching

This first criterion was suggested by Niklaus Wirth [25]. Wirth points out that widely used languages are not necessarily the best languages for teaching.

> *The choice of a language for teaching, based on its widespread acceptance and availability, together with the fact that the language most widely taught is therefore going to be the one most widely used, forms the safest recipe for stagnation in a subject of such profound pedagogical influence. I consider it therefore well worth-while to make an effort to break this vicious circle.*

It is interesting that Wirth was able to break this cycle for almost twenty years, but how easily we have reverted to use of commercial languages for the same reasons.

This criterion is echoed by Guido van Rossum [23].

> *...code that is as understandable as plain English.*
>
> *...reads like pseudo-code.*
>
> *...easy to learn, read, and use, yet powerful enough to illustrate essential aspects of programming languages and software engineering.*

Bertrand Meyer also suggests this criterion [16].

> *In some other languages, before you can produce any result, you must include some magic formula which you don't understand, such as the famous public static void main (string [] args). A good teaching language should be unobtrusive, enabling students to devote their efforts to learning the concepts, not a syntax.*

☑ To meet this criterion the language should have been designed with teaching in mind. The language will have a simple syntax and natural semantics, avoiding cryptic symbols, abbreviations and other sources of confusion. Associated tools should be easy to use.

2.1.2 The language can be used to apply physical analogies

This criterion was suggested by Seymour Papert [17]. Papert believed physical analogies involve students in their learning.

> *Without this benefit [using students' physical skills], seeking to "motivate" a scientific idea by drawing an analogy with a physical activity could easily denigrate into another example of "teacher's double talk".*

This idea is extended to "microworlds", a small, simple, bounded environment allowing exploration in a finite world.

☑ To meet this criterion a language would need to provide multimedia capabilities without extension. Perhaps more critical is the effort needed to get students to a stage where they could access this potential and how consistently it is applicable across environments (say between operating systems).

2.1.3 The language offers a general framework

The primary goal of any introductory programming course is to introduce students to programming. As such, the language itself is not the focus of instruction and any skills learned in one language should be transferable to other common languages. Bertrand Meyer suggests the following philosophy [16].

> *A software engineer must be multi-lingual and in fact able to learn new languages regularly; but the first language you learn is critical since it can open or close your mind forever.*

☑ To meet this criterion the language should make it possible to learn the fundaments and principles of programming, which would serve as an excellent basis for learning other programming languages later on.

2.1.4 The language promotes a new approach for teaching software

In an introductory course, language is but one part of the learning for a novice. It may be valuable where a language itself and associated tools can assist in learning to apply the language. Bertrand Meyer [16] suggests an introductory 'programming language' should be...

> *...not just a programming language but a method whose primary aim — beyond expressing algorithms for the computer — is [to] support thinking about problems and their solutions.*

☑ To meet this criterion the 'language' should not only be restricted to implementation, but cover many aspects of the software development process. The 'language' should be designed as an entire methodology for constructing software based on 1) a language and 2) a set of principles, tools and libraries.

2.2 Design and Environment

The following criteria describe the aspects of the language that relate to design and the environment in which the language can be used.

[1] http://www.papert.org/

[2] http://www.cs.inf.ethz.ch/~wirth/

[3] http://www.python.org/~guido/

[4] http://se.ethz.ch/~meyer/

2.2.1 The language is interactive and facilitates rapid code development

The potential to apply new programming ideas without requiring the context of a full program is valuable to novices [17]. The possibility to quickly start writing (and understanding) simple programs motivates and inspires [23].

- ☑ To meet this criterion the language and environments supporting its use should allow novices to implement newly acquired ideas, without having to establish the context of a full application. The language environment should provide students with interactive and immediate feedback on their progress.

2.2.2 The language promotes writing correct programs

The language Eiffel implements "Design by Contract" – a set of concepts tied to both the language and the method [14]. The aim is to move away from the prevalent "trial and error" approach to software construction.

> *By equipping classes with preconditions, postconditions and class invariants, we let students use a much more systematic approach than is currently the norm, and prepare them to become successful professional developers able to deliver bug-free systems.*

- ☑ To meet this criterion, students should be given ways to ensure that the code they write is correct and does not contain bugs.

2.2.3 The language allows problems to be solved in "bite-sized chunks"

It is desirable for any programmer to be able to focus on one aspect of a problem before moving onto the next. A language which supports problem decomposition is desirable as suggested by Papert [17].

> *It is possible to build a large intellectual system without ever making a step that cannot be comprehended. And building a system with a hierarchical structure makes it possible to grasp the system as a whole, that is to say, to see the system "viewed from the top".*

- ☑ To meet this criterion the language should support modularization, in functions, procedures or equivalent divisions.

2.2.4 The language provides a seamless development environment

When a novice begins to program it is valuable to understand the process that takes their program source to an executable program. Some Integrated Developments Environments can hide these details, obscuring this process for the sake of simplicity and rapidity which may be advantageous for an expert but less so for a novice. Other environments can assist in bridging the gap between design and implementation by, for example, converting architectural diagrams to code, and possible also reversing this process. Meyer suggests a "seamless development" can aid novices [16]. Such a language…

> *...enables us to teach a seamless approach that extends across the software lifecycle, from analysis and design to implementation and maintenance.*

- ☑ To meet this criterion the development environment should have an intuitive GUI for design and implementation which provides access to features and libraries, both for basic and advanced programming.

2.3 Support and Availability

The following criteria describe the support community and availability of the language and resources to teach the language.

2.3.1 The language has a supportive user community

Whether a language is a commercial creation or an open source project, its longevity will depend on the support for that language in the wider programming community. Where support is limited, resources and support may be a restriction for instructors and students. This criterion is suggested in [23].

- ☑ To meet this criterion, there must be sufficient support for students, faculty and others interested in learning and using the language. This support can come in different forms, such as web pages, course books, tutorials, exercises, documentation and mailing lists.

2.3.2 The language is open source, so anyone can contribute to its development

One of the benefits of an open source software project is the reduction of cost. Another benefit of an open source project is interoperability – where a commercial venture may seek to avoid compatibility with other systems to create a reliance on their creation. Beyond requiring a standard on which the language is based, this criterion seeks to differentiate languages whose development is the collaborative product of individuals. This criterion is suggested in [23] and continues over the following two criteria, even though the following criteria may also be applicable to languages outside the open source world.

- ☑ To meet this criterion the language should be the invention of a group who do not seek to create a commercial product and to which anyone can contribute if they wish.

2.3.3 The language is consistently supported across environments

Programming is conducted within different operating systems and on different machines. It is useful to be able to offer tools to students, which can be used in many environments rather than just one. This gives access to students regardless of location or setting.

- ☑ To meet this criterion the language should be available under various platforms.

2.3.4 The language is freely and easily available

For students, cost can be a significant issue. Students who are unlikely to continue programming beyond an introductory exposure will see little return from an expensive language or IDE.

- ☑ To meet this criterion the language should be free from subscription or obligation and available worldwide without restriction.

2.3.5 The language is supported with good teaching material

It is beneficial for both instructors and students if teaching materials are available for a particular programming language.

These can provide alternate perspectives and suggest appropriate curricula. This criterion was suggested by Meyer in [16].

- ☑ To meet this criterion, current textbooks and other materials should be available for use in the classroom.

2.4 Beyond Introductory Programming

The following criteria describe considerations beyond an introductory course. It is useful to consider how well a language can be used into various levels of a computing degree program as learning new languages, although valuable, can be a costly exercise if every new course requires its own language. As such an introductory language should also be examined from the perspectives of advanced levels of undergraduate programming and the programming industry. Moreover, using a language which is applicable in other contexts beyond introductory programming allows students to explore real world application domains using a powerful language and environment. This is especially relevant to students who wish to learn more, or at a faster pace.

2.4.1 The language is not only used in education

Students may be more motivated by a language that is not simply used within an educational setting. This criterion was suggested by van Rossum in [23].

> *...suitable for teaching purposes, without being a "toy" language: it is very popular with computer professionals as a rapid application development language.*

- ☑ To meet this criterion the language should also be relevant in areas other than education, e.g. in industry, and be suitable for developing large real world applications.

2.4.2 The language is extensible

Novices may not be expected to write extension modules within an introductory programming course, but using existing language extensions has potential to make the learning experience more motivating and exciting. Using modules, teachers can tailor tuition according to the interests of the students, allowing them to accomplish more than with the base language alone. Extensibility also allows a language to be applied to a larger variety of problems later in learning and professional use. This criterion is suggested in [23].

- ☑ To meet this criterion a language, which can be effectively used with only a small integral subset of features, should make it easy to access advanced functionality that is not directly accessible.

2.4.3 The language is reliable and efficient

Compilation speeds no longer seem to be as much of an issue as they were in 1971, when Wirth announced Pascal [25], however the ease with which a novice can take their source and produce an executable is still relevant.

> *...dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any nontrivial system is bound to contain mistakes forever.*

This is balanced by the need to involve the novice in this process and facilitate debugging.

Speed of execution still differentiates some languages, for example a distinction can be made between the products of the procedural and functional paradigms because of how closely each relates to the model execution used by processors. It could be argued that novice programmers rarely use the potential for speed in a language; however it could equally be argued that an academic setting is the perfect place to explore such limits.

It almost seems unforgivable that any compiler or environment for programming could be, in itself, flawed. Perhaps with modern 'bloated' industry languages, complexity within monumental libraries can bewilder novices.

From a pedagogical perspective, this criterion has a low priority in relation to other criteria. However, the decisions made by instructors are never purely motivation by pedagogy.

- ☑ To meet this criterion the language must be useful in creating high speed applications.

2.4.4 The language is not an example of the QWERTY phenomena

Papert suggests some languages continue to be used because of historical reasons and the justification for this continuation is often manufactured [17]. He defines the QWERTY phenomena in relation to the QWERTY keyboard.

> *There is a tendency for the first usable, but still primitive, product of a new technology to dig itself in.*

- ☑ To fulfill this criterion the language must show its usefulness now and into the future beyond its applicability in the past.

3. LANGUAGE COMPARISON

With criteria given it is possible to compare languages in an objective fashion. It should be stated that construction of such criteria suggests the biases of the authors of this paper. By choosing other inspirational figures, another set of criteria could have emerged and even within the works of the prominent figures chosen here, new criteria could have been promoted and others given less prominence. Also, any application of these criteria is subject to the authors' judgment.

Not all criteria can be applied equally to each language. For instance some languages are defined as a standard which is implemented by multiple groups in the form of compilers. Such a language may or may not be accompanied by additional tools in its delivered form. Other languages are developed by one group only and delivered with a specific set of tools. For this reason it is often difficult to judge if a criterion is applicable to a particular language and to what extent additional tools should be considered as part of the 'language'. For this reason, several notes have been added with the comparison.

The languages chosen in this comparison are chosen because they were in use during the most recent Census shown in [6] and are known to be in current use. This Australian/New Zealand source is a comprehensive survey of languages currently used in introductory programming. The inclusion of other languages could also be argued.

In this comparison all criteria are equally weighted, but the ordering presented here could easily be changed if criteria weightings were applied. All features are hardly equally important to all educators in all education situations.

Compared to the afore-mentioned feature lists given by Milbrandt [20] and McIver and Conway [15], the enumeration presented in this paper is more extensive. Features related to learning, design and environments have been considered previously, but including criteria concerning support, availability and possibilities beyond introductory programming seem to be unique to this study.

Some of the languages compared here can be regarded as "non-traditional" to introductory programming and might be avoided by some educators. Lack of strict typing in some languages (e.g. Python and JavaScript) is of concern to some educators. Some argue that teaching a language that is removed from a full industry relevant language can disadvantage students. Introducing programming using a simple language may cause students to run into a wall when having to deal with a more complex one later on. However, Mannila *et al* suggest students are not disadvantaged by having learned to program in a simple language when moving on to a more complex one [13].

For many instructors the choice of paradigm is primary and the language used must fall into a paradigm. There is as much literature discussing the value of teaching within one paradigm or another as literature discussing language choice (for example [7, 8, 9, 10, 12, 19, 24]). Certainly, if such an approach is necessary, then the results presented here must be qualified according to the instructor's choice of paradigm.

This given, a comparison has been attempted by the authors and the results are shown in Table 2. By this comparison, three languages are arguably the most suitable languages of those compared. Python and Eiffel rate highest which justifies their design as teaching languages. These are closely followed by Java, which is commonly associated with industrial applications. Other evaluated languages rated lower.

The comparison given is limited by the authors' experience with each language. The authors encourage all readers to consider how they would rate these languages, or perhaps others, according to the criteria.

4. CONCLUSIONS AND RECOMMENDATIONS

Not surprisingly, the authors' comparison suggests that the most suitable languages for teaching, Python and Eiffel, are languages that have been designed with teaching in mind. However this study also showed that Java, which is designed primarily for commercial application, has merit when considered as a teaching language.

By providing well founded criteria, this study has attempted to

Table 2: Languages Compared by Features

	C	C++	Eiffel	Haskell	Java	JavaScript	Logo	Pascal	Python	Scheme	VB
Learning											
Is suitable for teaching (§2.1.1)			✓				✓	✓	✓		
Can be used to apply physical analogies (§2.1.2)			✓		✓	✓	✓		✓		✓
Offers a general framework (§2.1.3)	✓	✓	✓		✓	✓		✓	✓		✓
Promotes a design driven approach for teaching software (§2.1.4)			✓	✓	*1		✓			✓	
Design and Environment											
Is interactive and facilitates rapid code development (§2.2.1)				✓			✓		✓	✓	
Promotes writing correct programs (§2.2.2)		*2	✓		*2				*2		
Allows problems to be solved in "bite-sized chunks" (§2.2.3)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Provides a seamless development environment (§2.2.4)			✓		*1						
Support and Availability											
Has a supportive user community (§2.3.1)	✓	✓	✓	✓	✓	✓			✓	✓	✓
Is open source, so anyone can contribute to its development (§2.3.2)									✓		
Is consistently supported across environments (§2.3.3)	✓	✓	✓		✓	✓	✓	✓	✓	✓	
Is freely and easily available (§2.3.4)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Is supported with good teaching material (§2.3.5)		✓	✓		✓		✓	✓	✓	✓	✓
Beyond Introductory Programming											
Is not only used in education (§2.4.1)	✓	✓	✓		✓	✓			✓		✓
Is extensible (§2.4.2)	✓	✓	✓		✓				✓		✓
Is reliable and efficient (§2.4.3)	✓	✓	✓		✓	✓		✓	✓		✓
Is not an example of the QWERTY phenomena (§2.4.4)		✓	✓	✓	✓	✓	✓		✓	✓	✓
Authors' Score	8	11	15	6	14	9	9	7	15	8	9

*1 Possibly with some IDEs, e.g. BlueJ (http://www.bluej.org)
*2 Possibly with unit testing

provide objectivity into what has been, up to now, frequently an emotive argument. The value of this work is to provide the potential for a strong argument to those who seek to promote a language change in an introductory course or perhaps over an entire undergraduate degree program.

This work may also be useful to communities of developers attempting to produce better programming languages for future novices and experts.

Extensions of this study in future work may attempt to clarify the criteria presented here, perhaps extending the criteria for specific purposes. Other languages may also be compared using this framework and it may be shown that other languages are useful for introductory languages according to these or other criteria.

5. REFERENCES

[1] Andreae, P., Biddle, R., Dobbie, G., Gale, A., Miller, L., and Tempero, E., *Surprises in Teaching CS1 with Java (School of Mathematical and Computing Sciences, Technical Report CS-TR-98/9)*. 1998, Victoria University of Wellington: Wellington.

[2] Bergin, J. *Java-- GOOD, BAD, and NOT C++*. 2000 [cited 30th August, 2006]; Available from: http://csis.pace.edu/~bergin/Java/SomegoodthingsaboutJava.html.

[3] Biddle, R. and Tempero, E., *Java pitfalls for beginners.* ACM SIGCSE Bulletin, **30**(2), 1998, 48 - 52.

[4] Chandra, S.S. and Chandra, K., *A comparison of Java and C#.* Journal of Computing Sciences in Colleges, **20**(3), 2005, 238 - 254.

[5] de Raadt, M., Watson, R., and Toleman, M. Language Trends in Introductory Programming Courses. In *The Proceedings of Informing Science and IT Education Conference*. (Cork, Ireland, June 19-21, 2002). InformingScience.org, 2002, 329 - 337.

[6] de Raadt, M., Watson, R., and Toleman, M., *Introductory programming languages at Australian universities at the beginning of the twenty first century.* Journal of Research and Practice in Information Technology, **35**(3), 2003, 163-167.

[7] Decker, R. and Hirshfield, S. A case for, and an instance of, objects in CS1. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*. (Vancouver, B.C. Canada, October 18 - 22, 1992), 1992, 309-312.

[8] Decker, R. and Hirshfield, S. The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1. In *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education*. (Phoenix, Arkansas, United States, March 10 - 12, 1994). ACM Press, New York, NY, USA, 1994, 51 - 55.

[9] Hadjerrouit, S., *Java as first programming language: a critical evaluation.* ACM SIGCSE Bulletin, **30**(2), 1998, 43 - 47.

[10] Hadjerrouit, S. A constructivist approach to object-oriented design and programming. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*. (Cracow, Poland, June 27 - July 1, 1999), 1999, 171 - 174.

[11] Hitz, M. and Hudec, M. Modula-2 versus C++ as a first programming language--some empirical results. In *Papers of the 26th SISCSE technical symposium on Computer science education*. (Nashville, TN USA, March 2 - 4, 1995). ACM Press, New York, NY, USA, 1995, 317-321.

[12] Kölling, M., Koch, B., and Rosenberg, J. Requirements for a first year object-oriented teaching language. In *Papers of the 26th SISCSE technical symposium on Computer science education*. (Nashville, TN USA, March 2 - 4, 1995). ACM Press, New York, NY, USA, 1995, 173-177.

[13] Mannila, L., Peltomäki, M., and Salakoski, T., *What About a Simple Language? Analyzing the Difficulties in Learning to Program.* Computer Science Education, **16**(3), 2006, 211 - 228.

[14] Mayer, R.E., Dyck, J.L., and Vilberg, W., *Learning to Program and Learning to Think: What's the Connection?* Communications of the ACM, **29**(7), 1986, 605-610.

[15] McIver, L. and Conway, D. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of the 1996 international Conference on Software Engineering: Education and Practice (SE:EP '96)*. (Dunedin, New Zealand, January 24 - 27, 1996). IEEE Computer Society, 1996, 309 - 316.

[16] Meyer, B. Towards an Object-Oriented Curriculum. In *Proceedings of 11th international TOOLS conference*. (Santa Barbara, United States, August 1993). Prentice Hall 1993, 1993, 585 - 594.

[17] Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA, 1980.

[18] Parker, K.R., Chao, J.T., Ottaway, T.A., and J.Chang, *A Formal Language Selection Process for Introductory Programming Courses.* Journal of Information Technology Education, **5**, 2006, 133 - 151.

[19] Ramalingam, V. and Wiedenbeck, S. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the seventh workshop on Empirical studies of programmers*. (October 24 - 26, 1997, Alexandria, VA USA). ACM Press, New York, NY, USA, 1997, 124-139.

[20] Stephenson, C. *A report on high school computer science education in five U.S. states.* 2000 [cited 31st August, 2006]; Available from: www.holtsoft.com/chris/HSSurveyArt.pdf.

[21] Stroustrup, B., *Learning Standard C++ as a New Language.* The C/C++ Users Journal, **May**, 1999.

[22] Tharp, A.L. Selecting the "right" programming language. In *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education*. (Indianapolis, Indiana, United States). ACM Press, 1982, 151 - 155.

[23] van Rossum, G. *"Computer Programming for Everybody." Proposal to the Corporation for National Research Initiatives*. 1999 [cited 25th October, 2006]; Available from: http://www.python.org/doc/essays/cp4e.html.

[24] Wallingford, E. Toward a first course based on object-oriented patterns. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*. (Philadelphia, PA USA, February 15 - 17, 1996). ACM Press, New York, NY, USA, 1996, 27-31.

[25] Wirth, N., *The programming language Pascal.* Acta Informatica, **1**, 1971, 35 - 63.

Student Perceptions of Reflections as an Aid to Learning

Arnold Pears,
Department of Information Technology
Uppsala University
Box 337
751 05 Uppsala, Sweden
Arnold.Pears@it.uu.se

Lars-Åke Larzon,
Department of Information Technology
Uppsala University
Box 337
751 05 Uppsala, Sweden
Lars-Ake.Larzon@it.uu.se

ABSTRACT

An important aspect in any learning situation is the approach that students take to learning. Studies in the 1980's built an increasingly convincing case for the existance of three learning approaches; deep, surface and achieving. These approaches are not mutually exclusive, and a single student may use any or all of them in combination. In addition, a connection has been demonstrated between deep learning approaches and understanding of the material being learned.

Encouraging deep learning behaviour, however, is a much more complex issue, since choice of learning approach seems to be dependent on the manner in which the student experiences the learning environment. This paper explores the use of reflections in the instructional design of two computing courses based on the text of the reflections and student feedback regarding the reflection exercise collected through surveys and interviews. Student's learning approaches are infered from a textual analysis of the reflection texts. Results describing student's perceptions of the utility of reflections as a learning tool are explored using interview data collected from students in one of the study cohorts.

1. INTRODUCTION

Some of the most influential work on tertiary student approaches to learning is that of Marton and Säljö [7]; subsequently extended and complemented by studies of Entwistle [3], Biggs [1, 2] and Ramsden [8]. These works argued for the existance of two, now well known, learning approaches; deep and surface learning. Subsequently a number of other studies have attempted to link learning activities and environments to student's pre-disposition to adopt deep vs surface strategies in their learning.

While the deep and surface classifications are certainly the best known in computing education circles, many studies of learning approaches also focus on other aspects of the learning experience. Fox [4] observes that mis-matches in student and teacher expectations of the structure of learning can result in student frustration and increases the likelihood that students adopt a surface learning approach. This is not surprising since, as Entwistle and Ramsden have observed, choice of learning strategy depends strongly on the context in which the learning takes place. It is not the case that a "clever" or "high achieving" student always avails themselves of a deep learning approach. This observation lead Biggs to propose a further learning approach, "achiever" [1], where pragmatism and context play a large role in the choice of learning approach used.

A common theme in the literature on student learning, and encouraging deep learning and learner autonomy, examines on the role of reflection. This can also be linked to the work of Schön [9] which in part deals with the role of reflection in professional education. A number of studies in higher education also report increased focus on deep learning in student cohorts that were encouraged to actively reflect on what they had learned [10].

Given this body of literature and the implication that reflection can motivate deep learning behaviour amongst learners introduction of curricula elements related to reflection seems potentially fruitfull. Having introduced reflection exercises in order to encourage greater student engagement and increased deep learning behaviour another question arises. What effect is this intervention actually having on the learning behaviour of students, and what do students think that these new exercises are about?

This paper reports on several aspects of student's use of, and perceptions in relation to, reflection exercises in two computing courses. We present results based on the student's own reflections, which reveal several distinctly different student approaches to writing reflections. Students were also interviewed and asked to discuss the reflection elements of the course. Our analysis of interview data focuses on student perceptions of the purpose and utility of the reflection exercises.

The paper is structured as follows. In the next section we describe the course designs, student cohorts and study methodology. Section 3 presents selected extracts from the data and develops arguments based on our observations. This is followed in section 4 by a discussion of the implications of the study for the use of reflections to encourage deep learning behaviour, as well as student's assessments of the usefulness of reflection as an element of their learning activity. Section 5 contains our conclusions and outlines areas of future work.

2. STUDY

2.1 Study Context

2.1.1 Cohort 1

Study group 1 consisted of 22 third year students in Science and Technology Studies (STS) at Uppsala University studying in the 6th semester of tertiary studies. The median age of the sample was 24 years. The youngest student was 22 and the oldest 25 years of age. Nine of the twenty two study participants were women.

The course undertaken by the cohort was *Distributed Information Systems*. Students are assessed continuously

through-out the course, and the exam was only mandatory for students that wanted to try for the highest course grade. The cross-disciplinary structure of the program results in a course package that covers several different disciplines. This affects the distributed information systems course in two ways:

- The typical STS student entering the course has only taken two previous computing courses - a basic programming course and a course on algorithms and data-bases.
- As students are expected to be able to take other advanced courses in computing after this course, it needs to cover a broad area.

The course consists of condensed variants of four other computing courses: *computer networking*, *operating systems*, *network and data security*, and *distributed systems*.

Learning outcomes in the course are assessed using a variety of approaches.

- A personal portfolio (mandatory), 5%
- Written reflections (up to 10), 4% per reflection
- Host a lecture (mandatory), 15%
- Oral presentation of an assigned topic (mandatory), 25%
- Self-evaluation, 10%
- Course evaluation, 5%

In the personal portfolio, the students present themselves, their background and what ambitions and goals they have in relation to the course. These were later revisited during a self-evaluation, at which point the personal interview took place. Each lecture is "hosted" by 1-2 students, meaning that they take notes that are made available to everyone. They also use 5 minutes at the beginning of the subsequent lecture to give a summary of what was covered last time, together with 1-2 questions directed to the teacher. Some of the lectures consist of 2-3 20-minute student presentations with subsequent discussion. In conjunction with each lecture students may choose to write a reflection in which they discuss what was important, interesting, uninteresting and hard to grasp.

2.1.2 Cohort 2

Study group 2 consisted of 42 computer science students in their second semester of tertiary studies. Prior tertiary course structure and assessment experiences had been entirely in the traditional Swedish model of lectures, obligatory practical work and final five hour written exam. The median age of sample was 21 years. The youngest student was 20 and the oldest 33 years of age. Two of the forty two study participants were women.

The course undertaken by the cohort was *Digital Technology and Computer Architecture*. This course is taught in a non-traditional manner, with grades being determined by participation in discussions and presentations. Non-verbal assessment in the form of written reports and personal reflections also contribute to the final grade.

The course material was divided into six topic modules. Possible grades in the course and associated result codes are unstatisfactory (U), pass (G) and distinction (VG). In order to obtain a pass grade of "G" students were required to participate fully in four of the six modules and complete four online reflections related to the content of those modules. A distinction or "very good" (VG) grade in the course required students to not only participate in five of the six modules, but also achieve a distinction in four of their group's written reports.

Each module consisted of an introductory lecture, two group discussion sessions, and a group seminar presentation. Seminar presentations covered a section of the textbook material. Each group was assigned seminar material to discuss and present to the rest of the class. After the classroom presentation the group had a week to prepare and upload a report on their seminar material to the course Wiki.

Attendance at lectures, discussion sessions and the presentations was noted in order to determine if students had actively participated in the given module. After the completion of each module those students who had taken the module were asked to write a 300 to 400 word reflection related to their learning in that module. There was no final examination. The intention was to both motivate and reward active participation in activities that we believe promote good learning outcomes.

2.2 Methodology

Systematic analysis of textual material can be conducted in a number of established research traditions. The approach in this study is influenced by qualitative approaches, in particular Phenomenography [6] and content, or textual, analysis [5]. In the current context content analysis is more relevant, since we are interested in structural characteristics in the nature of the reflections generated by the students, and in assessing the frequency of occurrence of different qualitative categories of reflection.

The reflections from Cohort 2 have been studied in order to identify distinct but characteristic ways in which students have engaged in the reflection process. We are particularly interested in exposing the manner in which students have approached the reflection exercise, as well as their perceptions of the purpose of reflections. This argument is built upon an analysis of the reflections themselves.

When exploring students' perceptions of the intent of the reflection exercises, as well as the utility of this type of written reflection as a self learning tool, we have used interview data from Cohort 1. Here the approach taken identifies common trends observed in the data.

2.3 Data Collection

2.3.1 Cohort 1

Student reflections were managed using a Wiki system, which allows users to add, remove or otherwise edit content quickly and easily. Thus, all reflections are available as web pages on the course homepage, but permissions have been set so that reflections can only be read by the student who wrote it and the teaching staff. For each reflection, there is an editing history which can be used to study the temporal aspects of how the reflection was produced. Moreover, it is possible to view an access log

for each reflection to study how students return to the reflection at different times throughout the course. Some instructions about the content and structure of a reflection were supplied on the course web page. The following quotation is an English translation of the original instructions in Swedish, the full instructions are reproduced in Appendix A.

> In a reflection you should reflect about a lecture or a mini-seminar.
>
> - Was there something that was especially interesting?
>   - If so: what and why?
>   - If not, why not?
> - Was there something that was confusing or unclear?
>   - If so: what?
>   - If not, what was the least clearly explained in what was covered?
> - Was there something that was totally irrelevant or felt meaningless?
>   - If so: what and why?
>   - If not, what was the least clearly explained in what was covered?
> - What was the most important thing you learned and why?

Halfway through the course, students were asked to volunteer to be interviewed about their progress and the assessment methods used in the course. A considerable fraction (73%) of the students were willing to assist us and were interviewed. In the interviews, which lasted 45-60 minutes each, a number of questions were asked about the different forms of assessment used. These interviews were recorded with permission from the students and parts of them transcribed. The script used to guide the interview is presented in Appendix B.

After the end of the course, students were invited to write an evaluation about the assessment techniques used in the course. In this written evaluation, one section was entirely about the usage of reflections - how they had been percieved and used throughout the course. There were also questions about students reaction to the idea that reflections can be used as a study technique in other courses. The questions asked in relation to reflections in that evaluation are given in Appendix C.

2.3.2 Cohort 2

Student reflections for this group were also collected using the the departmental Wiki system. The same editing history data and access permissions were used as for Cohort 1. What differs is the instructions given about the content and structure of a reflection. The instructions given to students regarding the reflections were as follows.

> Each reflection should comprise approximately 300-400 words dealing with the following aspects of the last phase of the course.
>
> - What was the phase/module was about?
> - What aspect surprised you most?
> - What is the most unclear part of the material?
> - Your impression of how well you understand the material, perhaps use a scale of 0-5 where 0 means understood nothing, 1 a little, 2 some, 3 at least half, 4 most, 5 all.

3. ANALYSIS

3.1 Student Perceptions

On the course web pages, both student cohorts were given instructions about how to produce their reflections. However, no motivation for including reflections in the assessment process was given. Consequently the students made their own assumptions about the instructor's motivations for including reflections among the assessment criteria.

In the evaluation questionaire data from Cohort 1, the following question was asked: "**What do you think the motivation for this part of the assessment was?**". To this question, a majority of the students responded that they felt that the primary motivation was to act as a feedback mechanism for the teaching staff. Those who stated a secondary motivation expressed it as a way to improve the learning process for the students.

This result was unexpected, and also slightly disturbing. Clearly, a majority of the students considered use of reflection exercises to be motivated by the needs of the *teachers* rather than those of the students. Useful as reflections might be for feedback purposes, the main instructor motivation for them was to have students think more about what they were taught to support deeper understanding of key topics. While deep learning processes might have been encouraged anyway it is interesting to note that this was not a goal shared or perceived by the majority of students.

This result can be contrasted with the results of the textual analysis of the data from Cohorts 1 and 2. Coding of the nature of the content of reflections recorded for both cohorts reveals three broad categories.

a) Those who fulfilled the formal requirements without engaging in significant deeper reflection (also few edits/revisions)

b) Those who spent time and effort on their reflections, often correlated to a larger number of revisions, and comments about relationships to other aspects of the material.

c) Those who used the reflection as a medium to communicate with the lecturer.

In the following discussion extracts are labled with fictitious names and the number refers to the cohort to which the student belonged. Gender has been preserved in the names for completeness, though we do not feel that this has a bearing on the present analysis.

Examples that typify the categories are found in the reflection texts of both cohorts. It is worth noting that by far the largest number of reflections are those of type (b), which leads us to conclude that reflections were a worthwhile technique in terms of encouraging reflective behaviour related to the subject matter to be learned.

An example of a type (a) reflection is that of Jones2:

> The first module covered the basic computer components; CPU, primary memory, secondary memory and I/O units. My group's area was secondary memory, primarily magnetic discs. Studying in groups helps to motivate me (and most others too I think), however one downside is you don't learn as much of the other groups' material as of your own. I will need to read more about the others' topics next module. Overall the material wasn't very complicated ("Basic" afterall).

Note that this type of reflection occured very seldom. The few examples are also related to the first module, where people were still getting used to the idea of writing regular reflections, or were made by students who did not complete the course.

A more interesting and personal approach to reflection is provided by Sam2 who says the following while reflecting on the content of the module that convered assembly programming. The quotation is translated from the Swedish original by the authors.

> "
>
> I knew beforehand that this module would be difficult because I understood that Assembler is not the easiest code to write. But, what I thought was the most problematic were the seminars held by the other students, I think it went very quickly, and here we really needed more time to go through Assembly. It felt like a hassle to be cast into programming in a language but on the other hand the practical work was very good. And, with the help of the lab assistant I understood more what it was all about. In this module I learned the most from the prac work and I guess that I understood more of the big picture after my prac partner and I had handed in everything.....
>
> What I thought that this was the most interesting in this module was the actual coding [of an assembly program] we did even though that was hard and we really had to work hard to see what mistakes we had made no and then, that we started to understand how close to the hardware and memory we really were. I learned to save a lot of time by planning on paper first and then try to create functions. If you first try to write the function on paper and see how it will work one avoids a lot of the problems and effort which otherwise needs to be put in [to make things work]. That was very useful to learn. Earlier one maybe hasn't structured the program or how the functions should work in advance. But, now we have learned that programming can go pretty well.
> "

Another reflection that has evidence of both reflection, identification of weakness in personal knowledge or understanding, as well as intent to connect to other areas is that of Andrew2, who says:

> " The content of the third module consisted of the Assembly Language level. The module described how the assembly language is implemented as a translation rather than interpretation, what the basic instructions and pseudo-instructions of an assembly language often are, and how it can be used in practice. The chapter described the format of the assembly language statements, common time complexity gains when rewriting high-language level code in assembly, macros and pseudo-instructions, the process of the assembly translation and the property and working scheme of the linking process and it's [sic] different ways (timing) of replacing virtual addresses into real addresses.
>
> The most surprising section of the chapter was the part about the relocation problem and how an object module is structured. Also new was the different times when the actual binding of symbolic addresses into absolute physical memory addresses can be made, with benefits depending on what system the code will be run upon.
>
> I didn't find any sections completely unclear or extremely difficult, though the part about windows DLL files was somewhat difficult at first.
>
> I think my understanding of the chapters in the module is equivalent to a 4 on the reflection page scale. I feel I understand most things, except perhaps for some parts in the dynamic linking sections. I need to study more in detail the process of linking and how object modules are interconnected.
>
> One thing to mention is I realized when I saw my attendance in the course manager for the third module (U) that I must have forgotten to write down my attendance on one of the lectures, I think it was the lecture on the 24th of April.
>
> Watching the presentations of the groups have been informative, and it gives me a feel for how I should prepare myself for my own presentation. Working on the assembly lab provides a good feel for the language, as well as a realization of how frustrating it can be sometimes :).
> "

Note that Andrew2, despite demonstrating several of the desirable characteristic categories of a reflection, also uses the reflection in the type (c) sense to communicate directly with the teacher. Towards the end of the course one of the modules did not run well. This stress prompted a number of students who had otherwise generated type (b) reflections to revert to a unique type (c) behaviour. One clear example of this is Jim2.

> " the lession & lecture the [date] was cancelled... why? If i should write about this module anyway, tell me "

However not all students reacted to the situation in this manner, and several wrote quite detailed reflections. Even in the later modules.

Another type of reflection that has been categorised as type (c) is typefied by a series of questions posed to the lecturer. A good example of this type of reflection is that of Luke1

> " ... Also I have a question about edge-chasing. If we have twenty processes which are all sitting waiting for each other we can solve that by one process choosing to kill its transaction, it is not so hard to understand that this opens up the chain for the others. But, who decides what transaction should die? Can it happen that all the processes decide to kill their transaction (more or less at the same time) and then there will be none left to execute. Then I wonder how often this happens in today's operating systems.....
> "

In summary the content analysis shows that all students who completed the courses generated at least one reflection with significant introspective content.

3.2 Time consumption

To the question *How much time did you normally spend on writing a reflection?*, students in Cohort 1 gave answers in the range of 15-90 minutes. Some also expressed that once they had learned how to write reflections, they could actually start writing it during, or directly after the lecture itself. Some students stated that they intentionally waited until the day after the lecture before writing the reflection, in order to be able to think more of the material before putting it down in writing.

Here it seems that students can be categorized into two groups from their answers: those who wanted to do a good reflection and get as much out of it as possible, and those who wanted to do a sufficiently good reflection as fast as possible. The first group typically spends 30-90 minutes on writing the reflection, while the second group spends only 15-30 minutes. Cross-referencing against the previous question regarding the motivation reveals that all the students that did their relections in less than 30 minutes on average assumed that the reflections are solely or mainly for the purpose of providing teachers with a feedback mechanism. There is also a stronger usage of the phrasing "*...being forced to...*" in the description of reflections among those who spent less time on the task.

3.3 Utility of Reflections

3.3.1 Feedback to students

For each reflection, the grading teacher commented on the reflection - especially on what the students had perceived most strange or ungraspable during the lecture. These comments were added at the end of the reflection together with its grading. In this way, students could return to read the comments. Although we could rely on the access logs to find out whether students actually did this, we also asked them *Have you read the feedback you've received on your reflections?*

On this questions, all students but one answered yes. Some of the student expressed it a positive way to get personal feedback to things they did not understand, and many appreciated the quality of the feedback. One student expressed that he/she mainly read the feedback to understand why he/she had not received the highest grade.

Clearly, feedback is something that is much appreciated. A few students have expressed it as useful to have this "*...personal, individualized communication channel with the teacher*" as a complement to asking questions during lecture hours.

3.3.2 Using the reflections

Students were asked whether they returned to the reflections they had produced (other than to just read the feedback) later throughout the course. On this question, it turned out about half of the students had returned to their reflections later in the course, in particular during the home exam and when preparing for the final exam. On a follow-up question on how useful they perceived their own reflection, several students stated that it was not the reflection in itself that was most useful, but rather the state of their mind that it represented. By reading their own text, they better remembered the rest of the lecture that was not covered in the reflection itself.

This way of using the reflection as a method to remember not only the material included in the reflection in itself, but also other things not documented in the text was a surprising usage of reflections that had not been foreseen. In retrospect, it is by no means surprising that a reflection can help its author to remember other things as well.

3.3.3 Student benefits

One interesting question is what benefits student perceive themselves to have obtained from producing reflections. There are several different answers from this, but the dominant one is that they perceive that they have learned more by having to think about what they heard during the lecture and what they really did not understand that well. The feedback from the teacher was appreciated as it tended to focus on the questions they had. Some students also value the exercise in producing constructive thoughts and questions.

When reading the answers, it seems like most students have appreciated the inner process involved in reflecting upon a lecture and putting it down in writing. Several answers claims that they perceive they have learned more than normally, and that they feel more alert throughout the lectures as well in order to get good material to reflect upon.

3.3.4 Reflections as a study technique

The final question about reflections that were given to students was *Would you consider using reflections in another course, even if you were not required to do it as a part of the examination?* All but one students were positive to this idea, but most of them also said that it could be hard to maintain the motivation if they did not have to do it. Two students clearly stated that they were going to try this out in other courses, and then particulary in courses that they perceived more difficult to understand. Although most students perceived a number of benefits from writing reflections in the previous question, few of them think that they would be able to maintain the motivation to produce them in a course where it was not required and motivated by a reward in the assessment.

4. DISCUSSION

There are clearly benefits to using reflections as a part of the instructional design of courses. In the course followed by Cohort 1 the combination of reflections with the personal portfolio means that one gets to learn quite a lot about the students. This includes information about their relevant background, how they perceive the same thing differently and what their weaknesses within the subject are. This knowledge about the different capabilities of the students is something that can be used in the classroom to increase the student activity. For instance you can use illustrative examples that relate to the personal background of some students and encourage them to share their experiences with the others.

The categories of reflections that emerge from the content analysis of Cohort 2 should not come as a particular surprise. However, what was surprising to us was that more than eighty percent of reflections written by both cohorts are of type (b). We interpret this to mean that many students do use reflections in the manner that we (the instructors) had intended, whatever their perceptions might have been. Many of the reflections contain references to discovery of lack of knowledge or insight and the need for further study, as well as attempts to link the subject matter of the reflection to other areas of the course material. We consider this to be evidence of deep learning strategies being deployed by these students.

Another clear benefit is the implicit feedback you get on your teaching. Given the questionaire responses and the outcomes of the content analysis it seems that this aspect of reflections does not pass unnoticed by students, so its "implicitness" can well be questioned. Feedback of this sort it is nonetheless useful. If a large number of the students are confused about a specific part of a lecture, you can return to that in the next one. Moreover, students seem more alert during the lectures and it feels comforting to know that some ratio of the students had made a serious attempt to absorb and review the material covered in the previous lecture.

There are significant instructor costs associated with using reflections. As students tend to produce quite a lot of the reflections early on in the course (in order to have done that so they can focus on other parts of the course), the workload is not evenly distributed. For the types of course structure reported here, one should be prepared to spend much more time giving feedback on reflections during the first half of the course.

A problem of a more technical nature has also emerged. As there was no support system for managing and grading reflections, legacy tools have been used. Students wrote their reflections in a Wiki, after which an email was sent to the instructor. The teacher then consulted the emails received, visited the Wiki pages that had been submitted, to which comments and the grades were then added. Grades were tracked of in a spreadsheet, and the overall grades were collated using a customized web based tool. Grading, consequently, involved using 4 different applications (Wiki, email, spreadsheet and an online learning management system) in the right way. To ensure all went well and no results were mislaid was a time-consuming process in which a support system for reflections would have been most useful. We intend to develop such a system during the fall in preparation for future courses.

5. CONCLUSION

The experiences from using reflections as a form of assessment in two computing courses are positive. Students perceive that they learn more from it, and that the time they spend to produce the reflections is reasonable. When returning to their own reflections during the course, the reflection can help students recollect not only things mentioned in the actual reflection, but also other things they did not document. Although most students are positive, few believe that they would use it as a study technique in another course, unless they had to.

From a teacher perspective, there are several positive benefits to be expected: you get a better picture of the students knowledge, more alert students during the lectures and the possibility to use examples that relate to students background. The problems in this course instance have mainly been an unevenly distributed workload in grading, plus the lack of a support system that ease the administration of reflections and their grading.

An analysis of the reflections themselves reveals that they appear to be used by the majority of students in a manner consistent with our expectations. That is, the students do indeed reflect on both their own actions and the material in manner that we believe acts to reinforce their learning and understanding. Coding for reflection that identifies lack of knowledge and linking to other areas of the material shows that a large number of students (over 80%) have demonstrated this in at least one reflection.

In general, reflections are heartily recommended as a form of assessment, and our evidence suggests that they encourage behaviours that have been linked to deep learning approaches in earlier studies. In doing so the workload distribution and availability of computer based support systems are important things to consider. Even in courses with relatively few students (40-50), grading reflections becomes a very time-consuming task without the right tools. For future courses, such a tool will be used, and we will also investigate the possibility of using peer assessment of reflections, in which the students would read each other's reflections.

Another dimension of future work includes studying how reflections as a form of assessment are perceived by different segments of the student population. Here we will try to determine if there are systematic differences in approach associated with particular programmes of study. One might also consider if there are observable gender differences in how students reflect.

6. REFERENCES

[1] J. Biggs. *"Student Approaches to Learning and Studying"*. Australian Council for Educational Research, Melbourne, Australia, 1987.

[2] J. Biggs. "Approaches to the enhancement of tertiary teaching". *Higher Education Research and Development*, 8(1):7–25, 1989.

[3] N. Entwistle. *"Styles of teaching and learning: an integrated outline of educational psychology"*. "John Wiley, Chichester", 1981.

[4] D. Fox. "Personal theories of teaching". *Studies in Higher Education*, 8(2), 1983.

[5] O. Holsti. *"Content Analysis for the Social Sciences and Humanities"*. Addison-Wesley, 1969.

[6] F. Marton and S. Booth. *Learning and Awareness.*

Mahwah NJ: Lawrence Erlbaum Ass, 1997.

[7] F. Marton and R. Säljö. "On Qualitative Differences in Learning". *British Journal of Educational Psychology*, 46:4–11, 115–127, 1976.

[8] P. Ramsden. *"Learning to Teach in Higher Education"*. London: Routledge, 1992.

[9] D. Schön. *"Educating the Reflective Practitioner"*. San Francisco, Josey Bass, 1987.

[10] F. Slack, M. Beer, G. Armitt, and S. Green. Assessment and learning outcomes: The evaluation of deep learning in an on-line course. *Journal of Information Technology Education*, 2:305–317, 2003.

APPENDIX

A. INSTRUCTIONS ON HOW TO PRODUCE A REFLECTION

In a reflection you should briefly reflect on a lecture or mini-seminar using the following outline:

- Was there something which was especially interesting?
  - If yes, what and why?
  - If not, why not?
-
  - If yes, what and why?
  - If not, what was the least clear in what was discussed?
- Was there something that seemed irrelevant or seemed pointless?
  - In that case, what and why?
  - If not: What was the least important thing that was discussed?
- What was the most important thing you learned, and why?

B. MANUSCRIPT FOR INTERVIEWS

Self assessment discussion DIS vt 2006

The following notes were used by the interviewer to structure the content of the course evaluation interviews conducted with students.

Introduction to the interview

- Is it OK to record this conversation?
- Please don't talk about this conversation with other students in the course before they have also had their interview.
- There are several reaasons for having this discussion:
  - to look at the initial questionaire and get some feedback on it,
  - have a quick look at the reflections,
  - revise the objectives of the course,
  - ask some questions related to a pedagogical study we are conducting,
  - look at the different types of approaches to teaching and working in a course,

Expectations

- What expectations did you have in relation to the course before it began?
- What were the sources of these expections?
- Why did you apply to enrol in this course?
- Would you like to reformulate your expectations?

Objectives

- What learning outcomes did you want to achieve through this course?
- What level of result are you aiming for?
- In relation to the porfolio material:
  - Is there any reason to adjust objectives?
  - Can I realise my goals, or do I need to adjust in some way?

Good and Bad

- What worked well in the course?
- What could be improved?
- Is there something that you want to point out or criticise?

Self Evaluation

- Draw a graph which represents the work you would normally put into a course.
- Draw a graph that represents how you have worked in this course so far.
- Draw a histogram of the work distribution of students in the course.
- Locate yourself on that histogram.
- How do you perceive/experience your work effort in comparison with other students?
- Are you happy with the level of your work put into this course?
- Do you need to change anything in order to achieve the personal goals that you have defined for this course?

Pedagogical Stuff

- If you were to guess what are the key principles on which this course is built what would you say they were?
- Let's talk about some key course elements.
- What do you think that the reflections have given you?
- How do you think the lecture host concept has worked;
  - for those that make the presentations?
  - for the audience?

Miscellaneous

- Is there anything else unique to the course that you would like to bring up;
  - the course structure?
  - the course content?
  - laboratories?

C. FINAL EVALUATION QUESTIONAIRE

After the course students were asked to help improve the course by filling out a questionaire related to the assessment and learning design that had been used. The section of that questionaire which was used to collect data presented in this paper was as follows.

- What do you think the purpose of this assessment component was?
- How much time did you spend on writing a reflection on average?

- Have you used the feedback you got on your reflections?
- Have you gone back and looked at your reflections? If so, when and why?
- Do you think that you will want to go back and read your reflections at some point in the future?
- What did you personally get out of writing reflections?
- Would you consider using reflections as a study approach in a course even if you were not "compelled" to do so by it being a part of the course assessment?

A Qualitative Analysis of Reflective and Defensive Student Responses in a Software Engineering and Design Course

Leslie Schwartzman
Department of Computer Science
Roosevelt University
Chicago, IL USA 60605
sla@acm.org

ABSTRACT

For students encountering difficult material, reflective practice is considered to play an important role in their learning. However, students in this situation sometimes do not behave reflectively, but in less productive and more problematic ways. This paper investigates how educators can recognize and analyze students' confusion, and determine whether students are responding reflectively or defensively. Qualitative data for the investigation comes from an upper-level undergraduate software engineering and design course that students invariably find quite challenging. A phenomenological analysis of the data, based on Heidegger's dynamic of rupture, provides useful insight to students' experience. A clearer understanding of the concepts presented in this paper should enable faculty to bring a more sophisticated analysis to student feedback, and lead to a more informed and productive interpretation by both instructor and administration.

Keywords

student feedback, confusion, learning, phenomenology, dynamic of rupture, defensiveness, reflectiveness

1. INTRODUCTION

The ideas of threshold concepts [5, 16] and reflective practice [6] have begun to receive considerable attention in computing education research; the former to describe challenges facing students in the discipline, the latter to describe work required of students to meet those challenges. The term 'threshold concept' is defined as a concept which, once grasped, leads to a transformed way of understanding or a new phenomenological awareness. Transition to the new understanding (enhanced - or new - mental or conceptual models [3]) or the new awareness is typically preceded by some period of difficulty [16].

In the literature, reflective practice, critical thinking, and learning are often associated. Plack and Greenberg [20] provide an overview, referring to a number of researchers well-known in the field: Kolb [15] links reflection to learning and describes critical thinking as taking time to revisit and process experiences from a number of different perspectives before drawing conclusions. Brookfield [9] links reflection to critical thinking. Critical thinking uses the analytic process of reflection to extract deeper meaning from experiences. Atkins and Murphy performed a meta-analysis of the many definitions of reflection found in the literature and identified an essential three-element sequence common to all [1]: A **trigger event**, typically an awareness of some uncomfortable (positive or negative) feelings and/or thoughts; then, a critical analysis of the feelings and thoughts, as well as the experience which gave rise to them; last, developing new perspectives as a result of this analysis. Many researchers note that while reflective practice is identified as important to learning, it has proven quite elusive to teach. Attempts to cultivate it often fall short [5,6].

1.1. Confusion, a part of Learning

Brown et al. [10] write of the confusion and uncertainty that attach to true learning, because it means encountering the unknown. Their comments indicate that the interval of difficulty which follows initial exposure to threshold concept material is filled with confusion, even when it is also filled with learning. Reflective practice is considered to play an important role in students' successfully navigating confusion. However, students in this situation sometimes do not behave reflectively, but in less productive and more problematic ways. Segal [22] has studied the relationship between reflective practice (or its lack) and the interval of confusion among adult learners. He asserts that, most of the time, exposure to challenging material (such as a threshold concept) initiates in the student an instance of Heidegger's dynamic of rupture, which culminates in reflectiveness or, alternatively, defensiveness.

Students' engagement in a non-reflective pattern has not received much attention in computing education research. Computing educators would benefit by a clearer understanding of the relationship between reflectiveness and its less desirable alternative, and the origins and indicators of each. This paper uses findings from the literature (particularly Segal's work) and experience with a course to investigate how educators can recognize and analyze students' confusion, and determine whether students are responding reflectively or defensively.

1.2. A more Sophisticated Approach to Qualitative Student Feedback: not just good and bad

Beyond student learning, the research has implications for understanding students' feedback, particularly their anonymous evaluations of the course and instructor. Applying Segal's analysis to this data not only enables better understanding and response to areas of students' difficulty. It also supports a more informed interpretation than simply 'yes' or 'no' votes on the course and instructor. A clearer understanding of the concepts presented in this paper should enable faculty to bring a more sophisticated analysis to student feedback, and lead to a more informed and productive interpretation by both instructor and administration.

Section 2 supplies motivation and background for the paper: how reflective practice matters to learning, the difficulty in cultivating it, and a preliminary look (the layperson's view) at reflectiveness and defensiveness. Section 3 holds a deeper exploration of reflectiveness and defensiveness, and their role in the dynamic of rupture. Section 4 describes CSX, a software

engineering and design course for upper-level undergraduates that students invariably find quite challenging, and from which qualitative data on student experience is drawn. Section 5 provides a representative sample of student feedback data, an explanation of the interpretation schemes used to analyze it, and preliminary analysis. Section 6, the conclusion, summarizes the relevance of Heidegger's dynamic of rupture - and Segal's analysis of it - for interpreting students' course evaluations, and looks to future work.

2. MOTIVATION AND BACKGROUND

Anonymous student evaluations in CSX consistently exhibit a bifurcated distribution: either strongly positive of the form "Good course, I learned a lot", or strongly negative of the form "Bad teaching, disorganized course". The combination suggests that something more complicated, not simply poor teaching, is happening. To investigate that possibility and elicit more meaning from students' anonymous evaluations and other qualitative feedback, this paper lays the foundation for applying a Heideggerian analysis (Segal's explanation of the dynamic of rupture) to the data.

2.1. Learning and Reflective Practice: an overview

Booth discusses the distinction between two broad categories in approaches to learning: surface and deep [5 p.145]. Surface approaches are associated with symbols or words. They direct attention on the sign, the representation itself. Students who use this kind of approach are more focused on the task per se, without consideration of its origins, consequences, or context. It can be summarized as "learning the text", and may take the form of literally memorializing course material.

In contrast, deep approaches focus on meaning, on that which is represented; it can be summarized as "learning through the text". Booth notes that deep approaches to learning are associated with quality learning outcomes, characterized by seeing the world in new ways [5 p.138,p.146], and understanding content from a multiplicity of critically different perspectives, a point also made by Berglund [4 p.19]. The capacity to shift among different perspectives to suit the task at hand has particular importance in computer science, where different understandings have relevance to each of many tasks, including designing software, writing code, or determining requirements with a user. Deep approaches depend on bringing students' behavior into their awareness and subjecting it to reflection so that *[their] meaning schemes may be transformed by reflection on anomalies* [16 p.13]. Counter-intuitive or threshold concepts (which form much of CSX content) are not learned in straightforward linear fashion, but rather require reflection [16 p.10].

Extending a metaphor from Plack and Greenberg [20 p.3], the kind of learning to which CSX is directed (as an advanced undergraduate course) comprises two main aspects, and can be likened to the double helix of DNA. One strand holds the cognitive content particular to computing and software development; it is acquired by cognitive effort, including memorization. One strand is composed of context, meaning, and their interplay; it is acquired through reflection, a practice common to all fields of learning, and related to the "deep learning" mentioned above. In her research, Booth draws on phenomenographic studies in multiple fields, because, as she notes, they all report similar findings with respect to learning, independent of content area [5 p.138]. Schon's book addresses multiple professions [21]. Similarly, for this paper I draw on literature from a number of professions, all directed toward the second strand of learning: that related to reflective practice.

2.2. Cultivating Reflective Practice

Three points frequently found in the literature with regard to cultivating reflective practice have particular relevance for this paper: First, the capacity for reflectiveness is founded in making explicit those factors which are typically left implicit, or making visible those assumptions which are typically taken for granted and unspoken [5, 22]. Second, some precipitating condition (Atkins et al.'s trigger event) gives rise to - and is required for - reflectiveness [20]. Third, significant difficulties attach to teaching and cultivating reflective practice [7]. Authentic reflective practice involves becoming aware of one's habitual behavior. Citing Nietzsche, Segal notes that if self-observation is done by rote, it leads to confusion rather than insight. '*Never to observe in order to observe. That gives a false perspective, leads to squinting and something false and exaggerated. … One must not eye oneself while having an experience; else the eye becomes an evil eye.' … [A] dogmatic commitment to observation produces a disengaged and decontextualised relationship to one's practice.* [22 p.75] citing and commenting on [18]. Authentic reflective practice is not done formulaically [20 p.1549]. Reflectiveness must come from a student's internal process, and questioning arises out of her dynamic engagement with the content. Booth also notes that questioning done by rote, or imposed by the teacher in a formulaic manner, leads to disastrous results [5 p.145].

2.3. Reflective vs. Defensive Responses: a preliminary look

From the layperson's perspective, reflectiveness and defensiveness are understood as radically different from each other. Defensiveness is associated with an incident-specific increase in emotion that overshadows other aspects of an encounter and effectively prevents further discussion of the topic at hand; in shorthand, an ad hoc reaction of: "NO, DON'T". Reflectiveness is associated with diminishment of emotional investment, a kind of long-term stepping back to see better; in shorthand, an attitude of: "Hmmm, I wonder..." Studies abound about the difficulty of engendering reflective practice [7]. These shorthand descriptions leave many questions unanswered, including: What makes reflectiveness so difficult to engender? What motivates it? How does defensiveness occur? Where does it originate? How can it be ameliorated? Both reflectiveness and defensiveness raise some difficult questions. The next section explores them more deeply, and lays the foundation to more precisely formulate and investigate the research question for this paper.

3. REFLECTIVENESS AND DEFENSIVENESS, COMPONENTS IN HEIDEGGER'S DYNAMIC OF RUPTURE

According to Segal, in an article meant to support teaching and learning in adult education, reflectiveness and defensiveness represent alternate paths through Heidegger's ***dynamic of rupture*** [22] citing [13]. The action of the dynamic is founded in Segal's observations of adult learners, informed by his knowledge of Heidegger. It takes the form of a three-step sequence: rupture, explicitness, response (either reflective or defensive). In this section, I draw on the literature to analyze the dynamic, and examine each of its steps in turn, as well as

several underlying concepts on which they're based. Then the origins of reflectiveness and defensiveness can be expanded and refined, clarifying the distinctions and similarities between them in order to better address the research question specified in section 3.4.

3.1. The Dynamic of Rupture - Explicitness - Response: overview and underlying concepts

This dynamic can be explained by an example from Dreyfus [12], familiar to anyone who's traveling internationally for the first time, perhaps to attend a conference: Each of us "knows" what particular distance to stand apart from an acquaintance when engaged in conversation. In general, we have no awareness of the specific distance, nor that it changes in proportion to the degree of our intimacy with the other person, nor that we have been socialized to it, nor even that we are doing it. This "**know-how**" resides in the **realm of the unseen taken-for-granted**. However, when we encounter conference host country natives who use a different conversational distance, we experience them as standing uncomfortably close (or uncomfortably far away), and **we suddenly become aware of our accustomed distance**. The discomfort thus becomes associated also with the emergence into visibility of our own behavior. According to Segal - and it is borne out by the student data from CSX - this discomfort is experienced either reflectively or defensively. Segal's explanation of distinct forms of differentness clarifies the two possibilities.

3.2. Distinct Forms of Differentness

Segal [22 p.76] citing Bauman [2 p.143] distinguishes between two kinds of differentness or otherness: the oppositional (in shorthand: 'enemy') and the unknown (in shorthand: 'stranger'). The oppositional is defined according to the same rules as we, but oppositely. Continuing the example of interpersonal conversational distance, the international traveler may respond: "The (host country) natives are standing the wrong distance away. How unrefined and uncivilized. What a collection of unrefined clods and uncivilized barbarians." Their differentness is thus defined in opposition: their 'wrong' vs. one's own 'correct' distance, their 'unrefined' vs. one's own 'refined' nature, their 'uncivilized' vs. one's own 'civilized' actions. **Defining the other in opposition, as 'enemy', confirms one's view of the world**: One's concept of the correct distance, and who decides it, remains untouched. Questioning of one's own or the other's behavior is not required; in fact, it has no place.

Enemies share common boundaries; although they oppose each other, they have a common appreciation of the rules in terms of which they meet each other ... ***[Enemies] function in the space of the existentially familiar****...* [22 p.76] quoting [2 p.145].

Alternatively, the unknown is defined according to unknown rules, or perhaps not defined at all. The international traveler may respond, "What is happening here?", and eventually, "What does this mean? Does it mean that I have an accustomed distance? If so, how did I learn it, what length is it measured at, and how do I figure it out? Does it mean that they have an accustomed distance? If so, how do I learn it, what length is it measured at, and how do I figure it out? Is my lack of local know-how making them uncomfortable? How long will it take to learn, what will I do in the meantime? ..." Segal calls this *enter[ing] a state of inarticulateness* [22 p.78]. **Recognizing the other as unknown, as 'stranger' evinces the inadequacy of our worldview**. This unmediated encounter with the unknown poses a considerable challenge. Questions - but no real answers - abound. Bauman refers to it as the '***anxiety of strangeness***':

Strangers have no established boundaries in common - not even terms of which they meet each other... ***[Strangers] give rise to the existentially unfamiliar*** *... [T]here are no ways of reading [such] a situation that can be taken for granted. ...* ***The anxiety of strangeness is experienced not only in the face of the stranger but in the face of strange and unfamiliar situations - in any situation in which we cannot assume our familiar ways of doing things.*** [22 p.76] quoting [2 p.145].

Enemies and friends, or enemies and oneself, represent two sides of the same coin. Strangers - or strange, unknown situations - represent a different coin altogether.

3.3. Beyond Rupture and Explicitness: reflectiveness and defensiveness

Segal notes that both cognitive and emotional elements are involved in the consequences of rupture and explicitness, because high emotional arousal, either anxiety or excitement, forms an integral part of being attentive. Segal uses the term **reflectiveness** to mean the process of examining (and thereby possibly changing) currently held beliefs. He uses the term **defensiveness** to mean a refusal to examine, and a rigid holding to - even idealizing, currently held beliefs. Note that this requires something having been made explicit, in order to hold to it. **Both reflectiveness and defensiveness (or dogmatism) are freighted with uncertainty and anxiety, and either can follow equally from explicitness.**

Defensiveness serves a protective purpose: It shields the responder from having to experience the shock of estrangement (and concommitant unease and uncertainty) from the everyday taken-for-granted context in which he encounters the world. **A defensive response to explicitness is characterized by recasting the unknown (strange) explicit as the known oppositional (enemy) explicit**. It enables the responder to remain in the realm of the known and the unquestioned, without being forced to examine it. **It is enacted through the mechanism of projection, displacing onto others the uncertainty and anxiety engendered, and blaming them for one's predicament;** for example the international traveler's first response.

Boud also speaks to the challenges of reflectiveness and the emotional elements involved, citing earlier researchers and *characteristics of reflection such as perplexity, hesitation, doubt [11], inner discomforts [8], or disorienting dilemmas [17]. …* ***reflection involves a focus on uncertainty, possibly without a known destination*** [6 p.15].

In summary, **rupture is required for explicitness; explicitness serves as a pre-condition to both reflectiveness and defensiveness**. Both reflectiveness and defensiveness arise from encounters with the unknown, and include significant affective components. A defensive response means avoiding the challenges of uncertainty and its affective components; a reflective response means taking on those challenges.

A point of terminology: Segal consistently uses the terms **rupture** and **explicitness** to signify a sequence of two stages in Heidegger's dynamic, where explicitness means a sudden, unbidden dawning of awareness. He introduces the terms **reflectiveness** and **defensiveneness** as **forms of explicitness**, but throughout the paper, he uses them to convey a different shade of meaning: a kind of living with - in tension, not

accommodation - an extant explicitness. Occasionally, he writes about them as distinct from explicitness: *explicitness can equally lead to defensiveness [or reflectiveness]* [segal p 88]. I have taken that distinction as fixed, and use the term **response** to signify a subsequent (third) stage in Heidegger's dynamic, a stage consisting of either reflectiveness or defensiveness.

3.4. The Research Question

The most educationally productive question becomes clear: How to engender a reflective response in every student under all conditions, or failing that, how to transform defensiveness into reflectiveness. Addressing it requires understanding sufficiently the nature and origins of defensiveness and reflectiveness, to recognize and distinguish between them. This paper lays the groundwork by addressing a preliminary question: **What in student feedback data evinces instance(s) of the dynamic of rupture, and how are reflective and defensive responses distinguished one from another?** To investigate this question, I bring the literature to bear on data from course CSX, which is introduced in the next section.

4. THE CSX COURSE

CSX, a software engineering and design course, is offered to upper-level undergraduates. The partial course description in this section sets a context for analyzing qualitative student feedback, the focus of this paper (a fuller description of the course is left to another paper). To the extent that space constraints allow, this section contains elements of the course relevant to student experience: content, structure, pedagogy, and format.

4.1. Course Overview and Structure

CSX is meant to teach software development fundamentals in a way that transcends software tools and languages, yet engages students in the actual practice of software, not just a theoretical or anecdotal exposition. In keeping with the principle of technology-independence, pencil and paper could suffice - although students may prefer to use a word processor and printer - for every assignment except the last. In keeping with the principle of engaging students in the actual practice of software, the group project - and the course - concludes with an assignment to write a correctly running program consistent with the documentation that was used as a design medium for it and the encompassing program family.

CSX is organized in three segments: two iterations linked by an intervening bridge. During the first iteration, students work on a series of individual 'design and development' assignments, motivated by two purposes: Each assignment is intended to make explicit some point(s) that play a significant role in software quality, but which are often left implicit in programming courses for a computer science degree; for example, subtle ambiguities in specification. Each assignment is also intended to identify and clarify some distinction(s) that play a significant role in software quality, but which are often not addressed directly in those same courses; for example, functionality vs. implementation. The second iteration is devoted to a group project with multiple assignments that reprise the content of iteration I, in a more challenging problem; for it, students also draw on each other as resources. Two or three assignments related to design for ease of change provide a bridge between the two iterations. The bridge covers possibilities for criteria used in modular decomposition, the design of module interfaces, and the implementation of designated modules in ways that support maximum flexibility.

4.2. Course Format and Pedagogy

Success in CSX requires mastery of several counter-intuitive concepts. To support students' authentic reflectiveness, course pedagogy is guided by the principle that they learn most when engaged with the material through their own questions. As Booth recommends, new concepts are introduced through homework assignments rather than lectures, and these assignments are given without classroom examples that students can simply adapt and use as a template [5 p.149]. Consequently, students come to the next class meeting *with a background of half-formulated queries and difficulties [when] ... their own worlds ... encompassed the field of the new concepts, and they had questions of their own at hand, grounded in their own enquiry* [5 p.149]. A similar approach was taken by engineering mathematics faculty at Chalmers University of Technology in Sweden [5] citing [14]. Homework assignments are not regarded as having exactly one correct answer, determined by the teacher, and students' submissions are not treated as mistakes, especially during the first iteration. The degree of students' efforts on an individual assignment is judged both by what they submit and by their participation in class discussion (in small classes, these discussions demand more than simply reciting text). Grading policy has changed over time. More recently, a student's first iteration scores are not counted in figuring the final grade, provided that she makes a serious effort on each individual assignment.

Elements of CSX classroom dynamic resemble the conversational classroom described by Waite et al. [24]. The first half of a class meeting is devoted to discussing assignments just submitted or being returned. Students' submissions are introduced (anonymously) as a foundation for collective exploration and analysis. Students may, and frequently do, identify their own ideas, or introduce new ones. Their (mis)conceptions often come to light while discussing a proposed solution and its implications. The implications can themselves be further examined, in keeping with Booth's dictum about a requirement for real learning: To become aware of their own learning, and variants in the ways a phenomenon may be experienced, students must subject their own work, and others', to scrutiny and reflectiveness [5 p.137]. The power of the course to effect student learning derives partly from using their own work (their 'mistakes') as subject matter; this holds their attention and begins with what has meaning to them, two pedagogically important considerations.

4.3. CSX Content

Course content draws from the work of David Parnas, where design has primacy of place. Rather than a series of software projects to be coded with little attention to how that is done, the course is constructed as a sequence of assignments meant to illustrate points of practice, and to give students *sufficient instruction in how to put the pieces [of software development] together* [23]. In order for students to concentrate on a particular aspect of development, rather than be distracted by the complexity of a problem's content, the content domain is chosen as the smallest problem that can bring that aspect of software development into focus. For some homework assignments, the content may appear simple, even trivial; but treatment of that content - what is intended for the students to learn - becomes both sophisticated and accessible. This section

includes some specific material taught in class (subsection 4.3.1) and one homework assignment with explanations for the interested educator (subsection 4.3.2).

4.3.1. A working definition for Good Quality in Software

The immaturity of computing as a profession is reflected in the absence of consensus on the definition of good quality in software. A working definition was devised for the course; good quality software is defined as satisfying conditions:

i. it provides the designated functionality
ii. it has a conceptually manageable form
iii. it can be modified with an amount of work proportional to the amount of change in functionality requested: a small change in functionality requires a small amount of work; and the modified program satisfies conditions i., ii., and iii.

4.3.2. Sample CSX assignment

assignment 1 (definition of the contact( ) *procedure is based – with permission - on a homework problem from William Farmer at McMaster University)*

Write a procedure called contact( ), to receive 6 integer arguments: ax, ay, ra, bx, by, rb representing two circles a and b on the plane. circle a has center at a point with coordinates (ax, ay) and radius ra, and similarly for circle b. contact( ) should be written to produce the following:

1 if all points of circle a lie within circle b
2 if all points of circle b lie within circle a
3 if circle a and circle b have points in common
4 if circle a and circle b have no points in common
5 if circle a equals circle b
6 if circle a and circle b lie tangent to each other

associated reading: Professional Responsibilities of Software Engineers [19]

sample student submissions: Due to space constraints, very few of the many variations are enumerated below. Note: '[..check_k]' denotes computations to determine whether circles satisfy conditions for category k. Two variations of [..check_3] follow the sample contact( ) procedures.

```
Procedure_W   {
 if ( [..check_6 ]..)  { [ output_6 ]; exit; }
 else
 {  if ( [..check_5 ]..)  { [ output_5 ]; exit; }
    else
    {  if ( [..check_4 ]..)  { [ output_4 ];  exit; }
       else
       {  if ( [..check_3 ]..)  { [ output_3 ]; exit; }
          else
          {  if ( [..check_2 ]..)  { [ output_2 ]; exit; }
             else
             { if ( [..check_1 ]..)  { [ output_1 ]; exit; } }
} } } } }
```

```
Procedure_Y   {
 if ( [..check_5 ] == true )  {
    condn_1 = true; condn_2 = true; condn_3 = true;
    condn_4 = false;  condn_6 = false;
    print  condn_1  condn_2  condn_3
    print  condn_4  condn_5  condn_6;
    exit;
 }
 if ( [..check_1 ] == true )    {
    condn_2 = false; condn_3 = true; condn_4 = false;
    condn_5 = false;  condn_6 = false;
    print  condn_1  condn_2  condn_3
    print  condn_4  condn_5  condn_6;
    exit;
 }  …
}
```

variants of [..check3], *do circles have points in common:*

variationA_check3:

```
 {  if ((ra <= rb) or (rb <= ra))  {  condn_3 = true;  } }
```

variationB_check3:

```
 { if ((cax == cbx) or (cax == cbx)) { condn_3 = true; }}
```

point(s) to be made explicit: Students make assumptions to resolve ambiguity (which was deliberately inserted into the assignment) usually without realizing it. The variation in assumptions leads to variation in output for the same input, which becomes clear during class discussion: e.g., What output is produced by procedures W and Y for circles that fit in no category?, or circles that fit in multiple categories? The amount of variation surprises students, because each had been convinced of the absolute correctness and irrefutable nature of his assumptions.

lesson(s) to be carried forward into course: Become aware of the assumptions you make - and verify their correctness (or not) with the client (in this case, the instructor) or others involved.

some relevant separation(s) of concerns:
- ambiguity vs. clarity of specification;
- problem description provided vs. assumptions made.

All assignments in iteration I use the same geometric problem domain. The many variations among students' submissions offer a rich set of possibilities for discussion and learning. Over the six weeks of that segment, '*lessons to be carried forward*' include, among others: the effects of flow control structure on ease of change, (un)soundness or (in)completeness of [..check_k] conditions, distinguishing between functionality and implementation, and distinguishing between content domain and computation domain. Through these apparently simple assignments, students get access to - and learn ways to address - the less obvious but critical problems and distinctions in software design and development.

5. STUDENT FEEDBACK DATA

In choosing data to include in this paper, I attempted to select as representative as possible a sample of views expressed. However, I did exercise bias for one criterion, clarity: From among multiple student responses stating the same opinion, the most articulate was selected.

5.1. Data Sources

Recorded qualitative student feedback data on CSX is collected from a variety of sources; the variety has expanded over the eight semesters the course has been taught. Each time a new instrument for qualitative data collection was introduced for one semester, it has been retained for all subsequent offerings, sometimes with modification.

From the first course offering, students' end-of-term anonymous evaluations were recorded; and some quiz or exam questions were directed toward qualitative measures (e.g., "What worked well in your group? If something did not, how would you suggest doing it differently?"). From the second offering forward, students were assigned to keep logs during the group

project, and end-of-term interviews with each student were instituted for evaluating student performance. The logs were kept for accountability purposes: each student recorded all communication with other group members, including dates and times, participants, and tasks accomplished; they contained very little qualitative data. End-of-term interviews were conducted to determine an individual student's contribution to the group project and her knowledge of the course material involved; initially, only occasional notes were taken and preserved.

For the last three semesters, end-of-term interviews were recorded (by hand) for later analysis. They provide a means to better understand students' learning experience in CSX, and to refine teaching accordingly. In the most recent offering, during class discussions on the group project, students often spoke about material that they were clearly wrestling with, or thinking deeply about. In order to obtain an account in their own words, I invited them to record these thoughts in their logs; the students began to call them journals. Sources are noted for each piece of data included in the next section.

5.2. End-of-term Recorded Data and its Interpretation

No student explicitly states that she experienced the dynamic of rupture, much less engaged in a reflective or defensive response. Therefore, conditions must be specified that establish a classification scheme for the student feedback data. (Note that from the vantage point of the research question in section 3.4, the ideal specifying conditions - which may or may not exist - would cleanly partition the data into two sets: one definitively expressing reflectiveness, one definitively expressing defensiveness.) The actual classification conditions were devised by reasoning from the data, in the context of findings from the literature.

5.2.1. End-of-term Feedback: indications of reflectiveness

As noted in section 1, reflective practice is required for deep learning, which is characterized by new ways of knowing [5]. Therefore, data which explicitly evinces real learning, a change in thinking, or a change in practice can be classified as definitively denoting reflectiveness. Examples of this include:

end-of-term interviews from spring, 2005: Question 5. Looking back over the course, does it appear different to you at the end of the semester than at the beginning or middle? If so, how?
(student_S7): I never knew another way of learning software development but to take that blind route. In this project, I'd thought the main focus was code. When we sat in the lab coding, and it wasn't working, I thought: there must be something to that module design document (I just happened to look at it while sitting in the lab). It said 'this invokes that' and we weren't doing it that way, and we were more focused on getting the code done. And I thought why did [instructor] give us [these three weeks of other assignments] before code, if it's all about code? Maybe it's not all about the code. ... With the [design in documentation already done], you just have to worry about the final step of coding it in [any] language. ...

end-of-term interviews from spring, 2005: Question 6. What will you take away with you from the course?
(student_S4): Analyzing problems, analyzing software, and ways to go about developing software. I used to code software offhand without going off and thinking about it [first]. This course really helped me to go off and think about it. I'm not afraid anymore to program, I know that. The real duty behind software development isn't code. Code equals a small percentage. Really: it's sitting down and really thinking it out.
Q: What do you mean, 'afraid to program'? Did you used to be?
A little bit
Q: Can you expand on that?
Like the [kwic index] program: if you think about how to proceed, it would get overwhelming, almost like, 'Where do you start?'
Q: And now you have an idea of where to start?
Yes, now: I don't think about program in terms of lines of code, how many functions. Problems don't seem as big as they used to, they're simplified. [Now,] I'd take a project, break it down to its core elements, and really focus on that...
Q: When you 'go off and think about it', what does 'think about it' mean?
What is the underlying problem, what underlying job needs to get done? Break down the problem into pieces, each piece has its own duty or task, functionality. Instead of a big, round ball, [it becomes] things more like blocks.

excerpted from spring, 2003, anonymous student evaluations:
(student_A9): This course is a great course. It is very intellectually demanding and academically challenging. I was thinking of suggesting this course be required for computer science, but I would not. I think this course is only appropriate for those who are seriously interested in software engineering. Should there be a 2nd course based on this course? Absolutely.

5.2.2. End-of-term Feedback: indications of defensiveness

According to Segal, defensiveness is evinced by casting the source of explicitness (i.e., the teacher or the course) as a source of problems. Examples of this include excerpts from spring, 2003, anonymous student evaluations:

(student_A1): I think this class was much more difficult than it had to be. ... My main concern was trying to interpret what was being asked, instead of learning the material. -- A separate point - we spent 2 periods going over the [kwic index program] - Why? Why the line by line analysis of the KWIC index program? This has little value - except to confuse and bewilder the class.

(student_A10): Could not ask questions and get a straight answer, answers were always left ambiguous. ... Gave no examples of personal experiences, homework assignments were changed during class and not a full understanding was given, never told us what she really wanted or expected, lecture was often not helpful in understanding material, I wouldn't take this course again, I wouldn't take this course if it wasn't required ...

(student_A12): ... It took me 3/4 of the course to understand the "purpose" of the course and the approach. Most of the time the instructor appeared to be unprepared and unorganized. I had the feeling of "drifting" and not going anyplace. ...

5.3. In-process Recorded Data and its Interpretation

Almost all the data collected at semester's end, a kind of after-the-fact reportage, fits into one of the two classifications. However, for most of the group project, data recorded in the midst of students' actual process (as entered several times per

week in their logs) does not satisfy either defining condition given in section 5.2. It does consistently display a heightened level of affect, even anxiety, even among students whose projects later turned out well.

5.3.1. In-process Feedback: indications of anxiety

Excerpted from fall, 2005, group project logs

(student_S1, week 1 of 5, after group meeting): ... It seems to me that we were not getting anywhere very quickly and this undertaking was larger than I previously had thought. What seems like such a straightforward assignment has become very complex ...

(student_S2, week 1 of 5, after group meeting): ... I'm a very calm and balanced person, and never really get stressed out about anything homework-wise because of the timeline I usually follow when I work. This project is already starting to stress me out because of the seeming lack of progress that we've gotten through so far. It seemed to me to be a fairly straightforward assignment at first, especially given the examples of the circles and the KWIC index, and I had hoped to hammer out a good outline to the [documents] within the first two sessions. We're nowhere near that yet. ... It feels like we're getting nothing done, and right now I don't necessarily know where to work next on my own. ...

These excerpted log entries confirm the relevance of Segal's analysis; students are experiencing the effects of explicitness. That is, they are experiencing a period of confusion after being exposed to threshold concept material, but before developing the corresponding mental or conceptual models or acquiring a new phenomenological awareness. The distinctions described in section 5.2 between reflective and defensive responses do not fit this data. More work is required to identify and develop the skills for analyzing stand-alone in-process data. For now, it may be analyzed retrospectively, in the context of end-of-term feedback. A retrospective interpretation scheme can be explained through the example of the international traveler.

5.3.2. Retrospective Interpretation: footprints

In our example of the international traveler, a threshold concept regarding the existence and length of accustomed conversational standing distance might be phrased as: "I have been socialized to use a set of conversational standing distances particular to my culture. People in other cultures are socialized to the set of distances particular to their respective cultures. In any encounter with others, I can include within the field of my attention an awareness of our standing distance, adjusting it if necessary, for as long as it takes for us each to feel at ease."

If, as a result of responding reflectively, the traveler can come to this concept, she will eventually manage encounters with host country natives relatively free of uncertainty and discomfort; and she will be equipped with this new awareness for all her subsequent travels. If, however, the traveler responds defensively, the heightened affect has little chance to subside except by the traveler's returning home without having integrated any learning. In subsequent journeys this traveler will likely continue to encounter the world at his original level of phenomenological awareness, and may well experience a repeat of the dynamic of rupture on the same terms as before. Note this means that the **nature of the traveler's response (reflective or defensive) can be discerned after the trip by whether or not her view of the world has changed.**

One can apply this same reasoning to interpret in-process CSX student data: Due to students' more highly charged state during response (of either type) to the dynamic of rupture, data collected in the midst of such experience may not offer clean delineations between reflectiveness and defensiveness. More information can be gleaned by comparing this data with semester's end reportage. If in-process data indicates a student's heightened affect with regard to elements of CSX content or goals, one looks to that student's semester-end data, and **examines the footprints**. If his view of those elements has changed in any significant way, one can conclude - retrospectively - that he was engaged in reflectiveness. And if not, then not.

6. CONCLUSIONS AND FUTURE WORK

Segal's explanation for Heidegger's dynamic of rupture offers a tool to analyze students' experience of learning challenging material, and the confusion it elicits. It is explained in section 3 through the example of an international traveler. Subsection 6.1 holds a summary explanation. Implications for interpreting students' qualitative feedback are found in 6.2. Subsection 6.3 enumerates some directions for future work.

6.1. Reflective and Defensive Responses

To summarize Segal's explanation: explicitness (the unavoidable - and unchosen - coming into awareness of some phenomenon previously outside of awareness) plays a significant role in real learning. Explicitness does not arise from a linear progression of events, but only as a result of rupture or disturbance, an unexpected encounter with the existentially unfamiliar, either persons or situations, that induce the anxiety of strangeness. In turn, it gives rise to either reflectiveness or defensiveness; these arise from encounters with the unknown, and include significant affective components. In contrast to a lay person's casual understanding (section 2.3), a defensive response means avoiding the challenges of uncertainty and its affective components; a reflective response means taking on those challenges. **Reflectiveness does not equal contemplation.**

6.2. Anonymous Student Evaluations: beyond good and bad

This paper illustrates the relevance of Heidegger's dynamic of rupture (as formulated by Segal) for analyzing students' learning and experience of CSX, and its role in enabling a more sophisticated and productive interpretation of their course evaluations. That combination makes a strong argument for the potential value of the dynamic to other instructors in other computing courses, particularly as an interpretive tool leading to more effective use of their students' feedback. For example:

If student evaluations for a course exhibit a bifurcated distribution (one portion quite positive, the other quite negative), it may well result from thoughtful teaching of difficult material, particularly if some students speak about the value of the course for their learning. The students' feedback may be taken to indicate their experiencing the effects of explicitness (in Segal's terms), corresponding to a period of confusion as part of their learning challenging concepts; some are responding reflectively and some defensively. If the instructor is attempting to present the difficult material of computer science and the students are encountering the challenges it poses, the department's support of that instructor

and her capacity to foster reflectiveness will benefit both students and the profession.

6.3. Future Work

This paper lays the foundation for future work in a variety of directions: course changes resulting from analysis with the dynamic; clarifying exactly what in CSX triggers the dynamic of rupture; refining analysis of students' in-process feedback data; and cultivation of reflective rather than defensive responses.

7. ACKNOWLEDGEMENTS

The author thanks the anonymous reviewers for their suggestions and Raymond Lister for his thoughtful comments and intellectual generosity.

8. REFERENCES

[1] Atkins, S., Murphy, K.: "Reflection: a review of the literature", Journal Adv. Nursing, v18, pp. 1188-1192, 1993

[2] Bauman, Z.: Thinking Sociologically, Oxford: Basil Blackwell, 1990

[3] Ben-Ari, M., Berglund, A., Booth, S., Holmboe, C.: "What Do We Mean by Theoretically Sound Research in Computer Science Education?", ACM SIGCSE Bulletin v36 (4), ITiCSE Conference Proceedings, panel session, Leeds, UK, June, 2004

[4] Berglund, A.: "On the Understanding of Computer Network Protocols", L.D. Dissertation, Department of Computer Systems, Uppsala University, Uppsala, Sweden, 2002

[5] Booth, S.: "On Phenomenography, Learning, and Teaching", Higher Education Research & Development, v16 (2), pp. 135-158, 1997

[6] Boud, D.: "Using Journal Writing to Enhance Reflective Practice", New Directions in Adult Continuing Education, v90, pp. 9-18, summer, 2001

[7] Boud, D., Walker, D.: "Promoting Reflection in Professional Courses: the challenge of context", Studies in Higher Education, v23 (2) pp. 191-206, June, 1998

[8] Brookfield, S.D.: Developing Critical Thinkers: challenging adults to explore alternative ways of thinking and acting, San Francisco, CA, Jossey-Bass, 1987

[9] Brookfield, S.D.: Becoming A Critically Reflective Teacher, San Francisco, CA, Jossey-Bass, 1995

[10] Brown, J., Collins, A., Duguid, P.: "Situated Cognition and the Culture of Learning", Educational Researcher, v18 (1), pp. 32-42, January-February, 2001

[11] Dewey, J.: How We Think: a restatement of the relation of reflective thinking to the educative process, Lexington, MA, D.C. Heath, 1933

[12] Dreyfus, H.: Being-in-the-world: A commentary on Heidegger's being and time, division 1. Massachusetts, MIT Press, 1993

[13] Heidegger, M.: Being and Time, Oxford: Basil Blackwell (1985)

[14] Jarner, S., Martinsson, M., Fant, C.-H.: "Mathematics Education Project at Chalmers University of Technology. Working report". Unpublished manuscript.

[15] Kolb, D.: Experiential Learning: experience as the source of learning and development, Englewood Cliffs, NJ, Prentice-Hall, 1984

[16] Meyer, L., Land, R.:" Threshold Concepts and Troublesome Knowledge: linkages to ways of thinking and practising within the disciplines", ETL Project Occasional Report 4 (referenced on 24 February, 2006 from http://www.ed.ac.uk/etl/publications.html)

[17] Mezirow, J.: "How Critical Reflection Triggers Transformative Learning", in Mezirow, J. (editor): Fostering Critical Reflection in Adulthood: a guide to transformative and emancipatory learning, San Francisco, CA, Jossey-Bass, 1990

[18] Nietzsche, F.: "Twilight of the Idols", in Karl, L., Hamalian, L. (editors): The Existential Mind: documents and fiction, Greenwich, CN, Facett Publications, 1974

[19] Parnas, D.: "The Professional Responsibilities of Software Engineers", Proceedings of IFIP World Congress 1994, Volume II, August, 1994, pp. 332-339

[20] Plack, M., Greenberg, L.: The Reflective Practitioner: reaching for excellence in practice", Pediatrics, v116 (16) pp.1546-52, December, 2005

[21] Schon, D.: Educating the Reflective Practitioner, San Francisco, CA, Jossey-Bass, 1987

[22] Segal, S.: "The Existential Conditions of Explicitness: an Heideggerian perspective", Studies in Continuing Education, v21 (1), pp. 73-89, May, 1999

[23] Spohrer, J., Soloway, E.: “Novice Mistakes: Are the Folks wisdoms correct?”, Communications of the ACM v29 (7), pp. 624-632, 1986

[24] Waite, W., Jackson, M., Diwan, A.: "The Conversational Classroom", ACM SIGCSE Bulletin v35 (1), SIGCSE Conference Proceedings, Reno, NV, March, 2003

The Distinctive Role of Lab Practical Classes in Computing Education

Simon
School of Design, Communication, and IT
University of Newcastle
Newcastle, Australia
simon@newcastle.edu.au

Michael de Raadt
Faculty of Sciences
University of Southern Queensland
Toowoomba, Australia
deraadt@usq.edu.au

Ken Sutton
Southern Institute of Technology
Invercargill
New Zealand
ken@clear.net.nz

Anne Venables
Victoria University
Melbourne City
Australia
Anne.Venables@vu.edu.au

ABSTRACT
As part of a wide-ranging phenomenographic study of computing teachers, we explored their varying understandings of the lab practical class and discovered four distinct categories of description of lab practicals. We consider which of these categories appear comparable with non-lecture classes in other disciplines, and which appear exclusive to computing. An awareness of this range of approaches to conducting practical lab classes will better enable academics to consider which is best suited to their own purposes when designing courses.

Keywords
Computing education, phenomenography, lab practical class

1. INTRODUCTION
Although some forecasters say that they are on the way out, lectures still appear to play a significant role in the academic teaching of most disciplines. The academic stands before a large group of students and talks to them, aided perhaps by a board, a slide presentation, or a choice of other props.

Then there are the other classes, typically with significantly smaller groups of students. In a discipline such as history, a tutorial is the venue for students to discuss aspects of the topics that were covered in recent lectures. In a discipline such as mathematics, a tutorial is where students practise the techniques that they have been shown in recent lectures. In a discipline such as chemistry, a lab is where students learn practical techniques and conduct experiments that supplement, rather than recapitulate, lecture material.

Many computing courses have lab sessions, too, though some academics call them workshops and others call them tutorials. What is the role of these classes? Are they like history tutorials, like mathematics tutorials, like chemistry labs? Or are they something different, unique to computing education?

As one aspect of a wide-ranging phenomenographic study of computing academics, their understandings of lab practicals were isolated and analysed. This analysis sheds new light on these classes, suggesting strongly that they have varying roles in computing education, some of which are unlike the roles of tutorials or lab classes in other academic disciplines.

1.1 Computing Lab Practicals Defined
The terminology of classes is not consistent among computing academics, so the term *lab practical classes* or *practicals* is used here to mean classes in a computing lab in which students work at computers to learn the use of a software tool, device, programming language, or similar, with tutors at hand to assist them in learning to use that tool. This is quite distinct from lectures, in which students sit and watch while a lecturer explains or demonstrates the material to be taught.

Azemi [3] describes an approach where lab practicals and lectures are combined within a computing course. This approach yielded positive feedback from students and faster learning was observed, albeit at the cost of significantly greater effort from instructors. Simon [9] describes a similar approach using VET (Vocational Education and Training) teaching: "While a university subject will typically be taught with lectures to the full class followed by labs or tutorials for groups of 20 or so students, all VET teaching takes place in classes of 20 or so students. Each class is like a combined lecture and tutorial, and there is no analogue of the university lecture." Approaches such as this, while clearly of interest, do not fall within the scope of this study.

1.2 The Phenomenographic Process
A phenomenographic study begins with interviews of a number of subjects. The interview transcripts are then analysed to discover different ways that the subjects understand the same phenomenon. It is the contention of phenomenography that for any phenomenon there is only a small number of possible understandings, which are called *categories of description*, and that the understanding of any individual will fit into one or more of these categories. It tends also to be the case that for a given phenomenon, the categories of description are hierarchical. Commonly, the understanding of the novice will generally fit into the simplest category. As people become more familiar with the phenomenon, they will often progress to higher-level understandings, which will generally still encompass those at the lower levels. In such a case, the highest level of understanding, which encompasses all of the lower levels, will be in some sense a true and complete understanding of the phenomenon.

As important as categories of description are *dimensions of variation*, individual aspects of the phenomenon in which a

variety of values are found. These values are not in themselves different ways of understanding the phenomenon, but it is generally the case that a category will be associated with a set of comparable values across a number of dimensions.

One approach to a phenomenographic analysis is to look for dimensions of variation and the distinct values within each dimension; then to see what different apparent understandings of the phenomenon emerge when the researchers combine, say, the low-level values of each dimension, then the medium-level values of each dimension, then the high-level values of each dimension.

Another approach is to start by eliciting the different categories of description, perhaps somewhat holistically, and then to observe which values of each dimension appear to correspond with each category.

A third approach, as described by Åkerlind [2] in her excellent walk-through of the phenomenographic process, is to cycle between considering the categories of description and the dimensions of variation.

Regardless of which approach is taken, it will involve many iterations, and its outcome can often be expressed in a table whose rows are the categories of description that have emerged, and whose columns are the dimensions of variation, showing which value each dimension displays for each category.

1.3 A Phenomenographic Study of Computing Academics

As expounded by Marton [10], phenomenography is a valued tool for qualitative research in the social sciences, but it is not yet widely used in computing education research.

In early 2006, Raymond Lister, Anders Berglund, Ilona Box, Chris Cope, and Arnold Pears conducted a workshop on Phenomenography in Computing Education Research (PhICER). The workshop was conducted immediately prior to the Eighth Australasian Computing Education Conference, and is described in overview in a paper accepted for the Ninth Australasian Computing Education Conference [9].

Prior to the workshop, each participant was required to read a number of papers on phenomenography in practice and its application in computing education, to interview at least one computing academic, following a fairly general and wide-ranging script, and to transcribe the interview.

Interviewees were asked to speak about just one course, perhaps the one that they most enjoyed teaching, and were encouraged to speak freely and at length. The first questions, intended to elicit their approach to learning, covered such things as what they want the students to learn in the course, whether they explicitly discuss links between these things and the profession they expect the students to take up, and what problems students have with the course.

Next they were asked what distinct ways they present learning material to students, such as lectures, tutorials, website, email, etc. For each **X** of these ways, they were then asked

- Is there a typical structure to your **X**'s? Why do you do it that way?
- Is there something distinctive about your **X**'s, compared with other **X**'s in the department/school?
- Do you expect students to do any preparation prior to **X**'s? How do you encourage this? Why do you think it is important that students do this preparation?
- Can you give an example of an **X** which was more effective than most? Why was it more effective?
- Can you give an example of an **X** which was less effective than most? Why was it less effective?
- Can you imagine an alternative approach to make your least effective **X** better? For example, you might restructure it or present it in a different format such as a lab or a tute.
- Do you think it is appropriate for students to talk among themselves as they do an **X**? Why? What opportunity do you provide to support this?
- What sorts of thing do you expect your students to be able to do when they finish an **X**?
- What are the main problems students have with your **X**'s?
- How do your **X**'s link with your other (non-**X**) presentations of learning material?

Interviewees were next asked what distinct ways they assessed their students, followed by a comparable bank of questions for each assessment method.

The goal of phenomenography is to elicit the full range of understandings, not to categorise differences between different subsets of the population, so no demographic details were collected. We do know that our interviewees included younger and older academics, male and female, from universities and technical institutes, from at least five countries (Australia, New Zealand, Finland, Ireland, USA); but nothing in our collected data indicates which is which.

The interview script was based closely on one used by Kutay and Lister in an earlier study [8]. Although there is some difference between the two scripts, there is also substantial overlap, and the interviews from that study were included with those specifically gathered for the PhICER workshop. In all, 25 transcripts, anonymised and identified by a code, were brought to the workshop as data.

The body of the workshop, which ran for two days, consisted of some formal instruction in phenomenography and a great deal of analysis of the transcripts. By the end of the first day, participants had formed four groups, each working on a different phenomenon to be found in the transcripts. Analysis continued for some time after the end of the workshop, and indeed still continues. The results are described in overview in the previously mentioned conference paper [9].

1.4 Exploring Lab Practical Classes

This paper presents in detail the results of one group which concentrated on the specific parts of the transcripts that deal with computing lab practical classes, as defined in section 1.1 above.

Of the 25 transcripts, only 10 made any reference to what we have called a lab practical class. Some referred to clearly non-practical classes such as classroom tutorials without computers, and some made little or no mention of any classes of this sort.

The question that we asked as we began our exploration of the transcripts is "What are the variations in lecturers' experience of laboratory practical sessions in IT?"

2. DIMENSIONS OF VARIATION

We opted to start our analysis by looking for dimensions of variation, feeling that this might be easier than trying immediately to elicit categories of description. Following our examination of those parts of the transcripts that deal with practical classes, three clear dimensions of variation emerged: the level of preparation expected of the students; the links with lectures or other means of presentation; and the extent to which students are responsible for their learning.

Several other candidate dimensions of variation were discarded, either because we could find too few interview excerpts to give them credence, or because there was little or no variation, with most or all of the excerpts illustrating the same value.

It is usual when presenting phenomenographic results to illustrate each value of dimension of variation with quotations from the transcripts. We believe that the dimensions and their values are reasonably self-explanatory, and have chosen to keep the illustrative quotations for section 3, where the different values of each dimension are combined to explain the more holistic categories of description.

2.1 Preparation Expected of the Student

One of the questions in the interview script asked how much preparation the academic expected students to do prior to any type of class. The responses to this question showed distinct variation in the amount of preparation that academics expect their students to undertake prior to a practical class; this dimension of variation had four values:

- none;
- reading;
- doing; and
- both reading and doing.

2.2 Links to Lecture or other Facets of the Course

Another interview question asked how each type of class linked to each other type of class. The responses gave rise to a second dimension of variation, the relationship between lab practicals and lectures or other means of teaching; again we found four values:

- none;
- show in lecture → do in practical;
- do in practical → show in lecture; and
- show in lecture → do in practical → apply in assignment

2.3 Student Responsibility for Learning

A third dimension of variation was not related to any specific interview question, but was teased out from everything that the respondents had to say about their lab practical classes. This dimension, with three values, perceives the level of student responsibility for learning in a practical class as being:

- low (responsibility lying predominantly with the teacher);
- moderate; or
- high (lying predominantly with the student).

3. CATEGORIES OF DESCRIPTION

Armed with these dimensions of variation, it was possible to identify four categories of description of computing practical classes. As novice phenomenographers, we initially thought of these as distinct understandings of lab practicals. As our own understanding has matured, assisted by feedback from the PhICER leaders, we have come to appreciate that our categories might more accurately be described as four different *approaches* to lab practicals in computing education. We address this distinction in the conclusion, and crave the indulgence of experienced phenomenographers if we use the phenomenographic lexicon a little too loosely between now and then.

Table 1 summarises our findings and illustrates how the dimensions of variation combine to produce the categories of description.

Within each category of description the dimensions of variation are exemplified by one or more quotes, identified by the codes of the transcripts from which they are drawn.

3.1 The Lab Practical as a Class where Students Acquire and Practise Skills Independent of Concepts Covered in Lectures and Assignments

In the first category of description, academics perceive the practical class as somewhat independent of lectures. While the lectures will deal with the theory component of the course, the practicals are where the students learn about, acquire, and practise specific skills that form an independent practical component.

Table 1: Categories of description of IT instructors' experience of practical classes

Categories of Description	*The practical class is understood by IT instructors as a learning environment where students…*	Dimension of Variation		
		preparation	**links with other classes**	**responsibility for learning**
	acquire and practise skills independent of lectures or texbooks	none	none	predominantly with teacher
	practise the skills taught in lectures or textbooks	reading	show in lectures, do in practicals	mixed
	refine skills, troubleshooting problems encountered while acquiring the skills	doing *or* reading and doing	do in practicals, show in lectures	predominantly with student
	apply skills acquired in the students' own time	reading and doing	show in lectures, apply in practicals	predominantly with student

Because of this independence from the lectures or textbooks, little or no preparation is required for these practicals. Students are not even required to do prior reading.

"There is no textbook that tells them what DreamWeaver is about. How do you learn about DreamWeaver unless you actually put your hands on and do it? They learn very quickly without reference to textbooks."[I1]

For obvious reasons, the links between practicals and other classes are essentially non-existent.

"There's nothing particular in the labs that reflects back on general lecture material. Because the labs are primarily focused on the Haskell language, it's obviously related to any Haskell lectures I give, which is early in the semester, so there's a kind of one-to-one correspondence there. But there's not a great deal of correspondence to the general material or conceptual material that's spread widely in [the course] because the labs are really focused on mainly learning a brand new programming paradigm, which is only one part of the whole course. So there's not a great deal of cross-linking."[L1]

In this category, the teacher tends to assume the primary responsibility for the learning experience, from which it often follows that the class is highly structured.

"I try to always have an amount of questions that will fill their lab sufficiently... Some concepts I'd make them do loads of different examples to really hammer home what's going on... The lab sheets start off with a couple of examples to get them going, and then a couple of easy questions to get things started... If you give them little problems initially it helps overcome the "I can't do this" fear that some students have... I check in on every lab, and if there is anything causing difficulty, I'll do my best to banish it straight away... I try to get in early and make sure there are no obstacles to learning." [M1]

3.2 The Lab Practical as a Class where Students Acquire and Practise Skills Taught in Lectures or Textbooks

In the second category of description, academics view the practical class as the means for students to put into practice the skills that they have been taught in the lecture or the textbook. The lectures, for example, will be used to teach and demonstrate a particular skill; then in the practical class, students will be given exercises in the application of that skill.

In this category, the academic tends to expect the student to spend some time preparing for the practical – at the very least, attending the lecture or reading the relevant part of the text.

"You [the student] ought to be prepared before you go into the lab, you ought to have read the lab sheet." [T2]

The link between lectures and practicals is stronger in this category.

"They link with the lectures in that we'll cover something in the lecture, or I'll say 'you can do this', and in the labs we'll see how to actually do it." [E4]

Responsibility for learning is no longer primarily the teacher's; instead the students take up some of that responsibility.

"It's possible that if they're under-prepared they don't get that much out of it. In other words, if they under-prepare they don't complete all of the exercises. The way I believe I've got this set of exercises for each lab, and they should be able to complete them in a two-hour period, I believe. If they don't, if they're under-prepared then they may finish them..." [L1]

3.3 The Lab Practical as a Class where Students Refine and Troubleshoot Skills Acquired in their own Time

In the third category of description, academics expect the students to do the bulk of the skill acquisition in their own time, and perceive the practical as a class in which students are provided with help on aspects of the work that they have found problematic.

In this category the student is expected to do significant preparation for the practical; or rather, to spend significant time working to acquire the skills in question, so that the practical can be a productive troubleshooting session.

"Well, I really like it if they do some themselves. Two things I expect beforehand. First of all... I encourage them... to work through the whole of the textbook so that when they come to the tutorials, they're just doing the exercises that I've set them. And, if possible, they can do the exercises before the tutorial; then they only need to come to the tutorial and ask about anything they had trouble with, and they can perhaps go home early." [E4]

While the link between lectures and practicals is essentially the same as the previously defined category, there is sometimes an additional inverse link, where problems that arise in the practical are resolved in the lecture.

"It was a mutiny. I had demonstrators coming back to me saying 'You have to change this lab, they are going nuts in there... It was as if the very use of the word 'recursion' terrified them... I had to salvage this case in the lecture. I dug up a few of the solutions that I had been provided with by students and showed them... and then a student would go 'That's recursion!' When they saw that, they seemed to realise 'Hang on, this is actually easier than we thought.'" [M1]

Responsibility for learning is now predominantly the student's.

"Some students will have done all the questions, and come in ready with their questions, the ones they had trouble with. Other students won't have done anything, and they'll start working... Everyone's working at their own speed, covering the material. Some students will do all the questions, some won't. It depends how much they're willing to do beforehand at home." [E4]

3.4 The Lab Practical as a Class where Students Apply Skills Acquired in their own Time

In the fourth category of description, the emphasis moves from acquiring the skills to applying those skills. The troubleshooting assistance is still provided, but in the context of applying the skills to a particular task such as a project or a major assignment.

As with the previous category, students are expected to acquire the skills in their own time (or perhaps in earlier practical sessions) so that this practical can be devoted to work on a

project. The practical is now of less importance than the prior work, and can indeed become optional.

> *"An hour a week of tutorial / computer laboratory. Not many turn up to that very often; although they're available, they normally do it in their own time." [E1]*

The link between lectures and practicals will still be the same in this category, but the link between practicals and assignments now becomes explicit.

> *"Well they're parallelled, completely. Each lecture refers directly to a tutorial, which refers directly to an assignment. So they're all linked, and it's very obvious what the links are... The tutorial is the glue, if you like, between the assignment and the lecture material. It relates directly to the implementation or transfer of the material presented in lectures to an assignment situation." [E3]*

Responsibility for learning is now almost entirely the student's, with the academic providing few or no instructions.

> *"…The following four [classes] are, as I say, basically one-liners, saying implement the philosophies and material from the [lectures]; for example, it might have been on help, it might have been on how to implement pop-up help in a web [page], so the tutorial might just say 'implement pop-up help in your assignment' And that's it; that's what the tutorial says... I'm trying to wean them off, as much as possible, specific instructions on how to do a particular job, and get* them *to think about how it should be done." [E3]*

4. CONCLUSIONS

Three dimensions of variation in IT academics' understandings of practical classes have emerged through phenomenographic study. Through analysis of those dimensions of variation four categories of description of the practical class have been identified.

4.1 Similarity with Prior Work

By way of validation, similarities with prior work were sought and found with ease. There is a good deal of recent research focusing on university academics' conceptions of and approaches to teaching, along with the impact upon student learning of these conceptions and approaches. In summarising several studies from multiple disciplines across differing institutions, Åkerlind [1] noted two striking commonalities in the key dimensions of meaning that teaching has for university teachers. The first dimension focused on the *"transmission of information to students or the development of conceptual understanding in students"* and the second focus was towards *"the teachers and their teaching strategies or the students and their learning and development."* Building upon earlier work by Kember [7], Åkerlind proposed four descriptions of experiences that resonate strongly with the 'responsibility for learning' dimension of variation in our study. She posited a four-valued hierarchical shift in focus:

- focus on knowledge transmission by the teacher;
- focus on teacher-student relations;
- focus on student engagement; and
- focus on student learning.

Two years earlier, McKenzie [11] had reflected that university teachers should aspire to using approaches focused on student learning since these experiences encourage students to take deeper approaches to their learning, approaches that are often associated with higher-quality learning outcomes.

In an earlier qualitative study of teaching and learning, Fox [6] delineated four personal theories of teaching, which have been paralleled by more recent studies such as that by Prosser *et al.* [12].

At Fox's lowest level, which he calls *transfer*, the student is seen as a container into which the discipline knowledge is to be poured. Our first category of description, in which the students are taught new skills that are independent of lecture material, seems reasonably consonant with this theory.

At his next level, *shaping*, the student is viewed as a raw material to be shaped into a finished product whose specification is couched in terms of the discipline knowledge. It is tempting to relate this to our second category, in which the lab practical is where students acquire and practise skills they have been shown in other classes such as lectures; but the link is perhaps a little tenuous.

In the third level Fox moves the focus from the content to the student. At this level, *travelling*, the discipline knowledge moves somewhat into the background, as the countryside of a journey; the teacher's task is to guide the student through this countryside, pointing out features of interest along the way. This ties in well with our third category, in which students already have much of the knowledge, but are still being guided in its correct use.

At the fourth level, *growing*, the student is seen as already full of knowledge, the teacher's task being to cultivate that knowledge in the student, weeding and fertilising as appropriate. This ties in well with our fourth category, in which students already have the knowledge and skills, and seek help only in occasional aspects of their application.

The categorisation emergent from our study clearly ties in quite well with earlier theories, thus giving some validation. At the same time, if this work is to be anything more than a replication of earlier studies, its distinct and novel aspects need to be exposed.

4.2 Differences from Prior Work

The novel outcome from this work is the indication that, while computing lab practical classes are generally thought of as somewhat uniform, there are in fact a number of diverse approaches that appear to tie in with equally diverse educational purposes. In addition, some of these approaches appear distinctive to computing education. This can perhaps be best explained by referring back to the non-lecture classes of other disciplines, as mentioned briefly in the introduction.

Tutorials in, say, a mathematics course are generally intended for students to practice skills and methods acquired in lectures and/or textbooks. They thus fall neatly into category 2.

Tutorials in many humanities courses are for discussion of the topics that have been presented in lectures and/or texts. If they are for practice at anything, it would be at analysis and argumentation. With that interpretation, they probably fall into category 1, a class where students acquire and practice skills independent of concepts covered in lectures. Alternatively, if analysis and argumentation are explicitly taught in lectures, these tutorials too would fall into category 2.

Lab classes in some of the physical sciences appear to fall into category 1. In chemistry, for example, theoretical aspects of the

discipline are taught in lectures. In the labs, students follow tightly defined procedures to learn new techniques, and either discover properties that have not been addressed in lectures (category 1) or confirm properties that have been covered in lectures (category 2).

We have had to search rather harder to find tutorial or lab classes that fall into category 3 or category 4. We have not yet been able to verify this in the literature of other disciplines, but pending a thorough investigation, it appears to us that these categories are more or less exclusive to the creative disciplines. In art, design, music, and architecture, for example, we would expect to find classes where students apply their creative skills, with a tutor on hand to guide and assist rather than to show the way. It seems, therefore, that the presence in computing education of classes where students refine and troubleshoot (category 3) or simply apply (category 4) skills that they have already acquired confirms the often-argued position that computing is as much a creative discipline as it is a scientific or systematic one.

4.3 Is it Phenomenography?

The findings presented here are both interesting and significant. There remains a question, though, as to whether they (yet) represent phenomenography.

Phenomenography is clearly and explicitly designed to elicit different ways of understanding a particular phenomenon. Cope [5] conducted a phenomenographic survey to discover students' understandings of an information system. The 'information system' is the same thing throughout the study; all that changes is how students understand it. Berglund [4] studied students' understandings of various network protocols. The protocols remain fixed, but different students have different conceptions of what they are and how they are used.

By contrast, the findings presented here have elicited variation in *approaches* to conducting lab practical classes. It is not that different academics have different understandings of the lab practical: as the people who create the classes for their course, they can be assumed to have a fairly complete understanding of those classes. Rather, different academics have different uses for the lab practicals, and thus run them in different ways with different sorts of goal.

The phenomenographic method was used to analyse interview transcripts gathered for a phenomenographic study, but did not result in a categorisation of different understandings of the static concept of a lab practical. What has emerged instead is different approaches to lab practical classes, each suited to different purposes. In this sense, what has been achieved is not pure phenomenography. Fortunately, we do not believe that this makes it any less valuable.

4.4 The Value of this Work

How can the computing education community benefit from this work? McKenzie [11] showed that when university teachers were able to discern critical aspects of variation within differing teaching strategies, they moved to more student focused ways of experiencing their teaching. Therefore an awareness of the different categories of practical class will better inform academics who are designing courses; they will be able to consider the categories and decide just where they intend their own work to lie.

This leads to another aspect of the difference between these findings and the standard expectations of phenomenography. Aligned with the hierarchical arrangement of phenomenography's categories of description is an understanding that the more inclusive categories are in some sense better, that they are an ideal to be aimed for. Cope [5] would presumably be happy if all of his students expressed the most inclusive understanding of information systems, and Berglund [4] would likewise rejoice if his students all expressed the most inclusive understanding of network protocols. If this were the case with computing practicals, all teachers should be aiming to design their practicals in accord with the fourth category presented here. To the contrary, it is important to recognise that the different categories represent different approaches used for different purposes, and to appreciate the value of this distinction. Faced with a hierarchical categorisation of approaches to lab practicals, it is the responsibility of academics to decide which category or combination of categories is most appropriate for their courses.

A good example of combining categories is E3, who in a single 12-week course progresses deliberately in approach from category 2 to category 4:

> *"the tutorials ... the first, about four, are actually structured formal tutorials: do this, do this, monkey see, monkey do ... you know, do this, open this up, use this tool, right-click this, type this in the box, in the wizard, enter this data ... very, very specific instructions. The next four are less specific, and are mainly concerned with integrating the concepts of the lectures into their assignment. And the final four are basically one-liners: 'Integrate the material in the lectures into your assignment, full stop.'" [E3]*

Although it is the job of phenomenography to describe rather than to recommend, in response to comments from the referees for this paper we suggest that some academics might perceive the lower levels of our categorisation as being more suited to beginning students and the upper levels as better suiting advanced students. We have not yet analysed our transcripts to see if they support this suggestion, so at this point it must remain completely hypothetical.

4.5 Future Directions

Future work could include an investigation of academics' differing perceptions of students that lead them to adopt the different approaches delineated by this study. Further exploration of the current transcripts might provide a first step in this direction; but it is possible that these transcripts are not sufficiently rich with regard to this particular question, and that to answer it properly will require a fresh study with questions designed for the purpose.

Further study, of both academics and students, might also result in firmer or clearer guidelines as to which approach to lab practicals is better suited to which circumstances.

5. ACKNOWLEDGMENTS

This study was supported by a Special Projects Grant from the ACM Special Interest Group in Computer Science Education (SIGCSE). The authors thank Raymond Lister, Anders Berglund, Ilona Box, Chris Cope, and Arnold Pears for thinking of and running the PhICER workshop; and fellow PhICER participants Chris Avram, Mat Bower, Angela Carbone, Bill Davey, Bernard Doyle, Sue Fitzgerald, Linda Mannila, Cat Kutay, Mia Peltomäki, Judy Sheard, Des Traynor, and Jodi Tutty for their transcripts and collaboration.

6. REFERENCES

[1] Åkerlind, G.S. A new dimension to understanding university teaching. *Teaching in Higher Education*, *9*, 3, 2004, 363-375.

[2] Åkerlind, G.S. *Phenomenographic methods: A case illustration*, in *Doing developmental phenomonography*, J. Bowden and P.Green, Editors. 2005, RMIT University Press: Melbourne, Victoria, Australia. p103-127.

[3] Azemi, A. *Teaching Computer Programming Courses in a Computer Laboratory Environment*. 1995 [cited March 3, 2006, 2006]; Available from: http://fie.engrng.pitt.edu/fie95/2a5/2a55/2a55.htm.

[4] Berglund, A. *Learning computer systems in a distributed project course: the what, why, how and where.* Acta Universitatis Upsaliensis, Uppsala, Sweden, 2005.

[5] Cope, C. Educationally critical aspects of a deep understanding of the concept of an information system. In *Proceedings of The Fourth Australasian Computing Education Conference (ACE2000)*. (Melbourne, Australia), 2000, 48-55.

[6] Fox, D. Personal Theories of Teaching. *Studies in Higher Education*, *8*, 2, 1983, 151-163.

[7] Kember, D. A reconceptualisation of the research into university academics' conceptions of teaching. *Learning and Instruction*, *7*, 3, 1997, 255-275.

[8] Kutay, C. and Lister, R. Up close and pedagogical: computing academics talk about teaching. *Australian Computer Science Communications*, *52*, 2006, 125-134.

[9] Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., Bower, M., Carbone, A., Davey, B., de Raadt, M, Doyle, B., Fitzgerald, S., Mannila, L., Kutay, C., Peltomäki, M., Sheard, J., Simon, Sutton, K., Traynor, D., Tutty, J., and Venables, A. Differing ways that computing academics understand teaching. In *Ninth Australasian Computing Education Conference (ACE2007)*. (Ballarat, Victoria, Australia, 29 January - 2 February 2007), 2007.

[10] Marton, F. Phenomenography − a research approach to investigating different understandings of reality. *Journal of Thought*, *21*, 1986, 28-49.

[11] McKenzie, J. Variation and relevance structures for university teachers' learning: Bringing about change in ways of experiencing teaching. *Research and Development in Higher Education*, *25*, 2002, 434-441.

[12] Prosser, M., Trigwell, K., and Taylor, P. A Phenomenographic Study of Academics' Conceptions of Science Learning and Teaching. *Learning and Instruction*, *4*, 1994, 217-231.

Most Common Courses of Specializations in Artificial Intelligence, Computer Systems, and Theory

Sami Surakka

Helsinki University of Technology
P.O. Box 5400
FI-02015 HUT, Finland

sami.surakka@hut.fi

ABSTRACT

The degree requirements of American institutions offering top-level graduate programs in computer science were analyzed. The sample size was 59 institutions for undergraduate and 32 for graduate programs. The purpose was to solve which courses were the most commonly offered in the specializations of Artificial Intelligence, Computer Systems, and Theory. The results can be useful to professors who are responsible for any of these three specializations.

Keywords

Advanced courses, content analysis, degree requirements, document analysis, graduate program

1. INTRODUCTION

Curriculum design is a complicated issue including many different points of view that often contradict each other. Joint international efforts such as Computing Curricula 2001 (CC2001) [5] have been carried out to build general frameworks for designing and comparing different computer science curricula. CC2001 is useful for designing introductory and intermediate studies but it is less useful for designing specializations because it is limited to undergraduate programs (p. 1) whereas specializations are more typical in graduate programs. Specializations were covered in the report only on a general level but no recommendations for specific specializations were provided.

A benchmarking study was conducted in order to approach this curriculum design problem. The degree requirements of American institutions offering top-level graduate programs in computer science were analyzed. The purpose was to solve which courses were the most commonly offered in certain specializations.

Institutions use different names for the organization of advanced courses: at least the names concentration, option, specialization, and track have been used. In the present paper, specialization is used as a common name for these concepts. According to the ERIC Thesaurus [4], specialization means "concentration of interest and effort, or restriction of function, to a particular aspect of some larger area of endeavor (such as a field of study, occupation, etc.)—also, the process of progressive differentiation of functions."

We found in our previous research [16] that the four most common specializations were Computer Systems, Theoretical Computer Science, Software Systems, and Artificial Intelligence. The main target of our previous paper was Software Systems. The present paper is limited to the *other* common specializations than Software Systems; that is, to Artificial Intelligence, Computer Systems, and Theoretical Computer Science. However, the name Theory is used instead of Theoretical Computer Science because Theory is a more common specialization name than Theoretical Computer Science.

First, it was analyzed which specializations were most commonly offered when the purpose was to assure the previous results [16]. Second, the main purpose of the present research was to solve which courses were the most commonly offered in these three specializations.

The results can be useful to professors who are responsible for any of these three specializations when they consider which courses are required in a specialization and which courses are more suitable for electives. One should note that the course requirements were analyzed only in extent necessary to classify courses but no detailed results about course contents are presented. In other words, the results of the present paper are not useful for the more detailed question: Which topics should be covered on some specific course.

The related work is presented in Section 2 and the research method in Section 3. Section 4 presents the results of the present research. Finally, the research is discussed in Section 5.

2. RELATED WORK

First, the definitions of the main concepts are presented. Second, other publications are presented.

According to Information Technology Vocabulary [10, p. 22], artificial intelligence is

> the branch of computer science devoted to developing data processing systems that perform functions normally associated with human intelligence, such as reasoning, learning, and self-improvement.

According to the same source (p. 8), computer system is "one or more computers, peripheral equipment, and software that perform data processing."

Surprisingly, no standard definition was found for theory, theory of computation, or theoretical computer science. Instead, three sections of the journal Theoretical Computer Science [17] are suitable for characterizing the area: (a) algorithms, automata, complexity and games, (b) logic, semantics and theory of programming, and (c) natural computing. The third section deals "with the theoretical issues in evolutionary computing, neural networks, molecular computing, and quantum computing."

CC2001 [5, p. 235] presents a list of advanced courses divided into thirteen areas. These areas are not called specializations but they are interesting for the present paper anyhow. Out of these thirteen areas, the following four are most relevant to the present paper: Algorithms and Complexity, Architecture and Organization, Operating Systems, and Intelligent Systems.

Computer Engineering 2004 (CE2004) [14] is a curriculum recommendation targeted at computer engineering programs. It

presents a list of eighteen areas of knowledge of which the following three are the most relevant to the present paper (p. 12): Computer Architecture and Organization, Computer Systems Engineering, and Operating Systems. An example list of elective courses is presented as well but these courses are not divided according to the areas of knowledge (p. 35).

MSIS 2000: Model Curriculum and Guidelines for Graduate Degree Programs in Information Systems [7] is relevant because it is targeted at graduate programs unlike the other recent ACM curricula recommendations. It is recommended that an IS graduate program offers a specialization called career track. In a career track, at least twelve units (four courses) should be required which may include practicum (pp. 12 and 14). What is particularly interesting, sixteen examples for career tracks such as Electronic Commerce are listed and four example courses are suggested for each career track (p. 13). Course descriptions were not provided but nevertheless, the abstraction level and target area from the viewpoint of degree structure are almost the same as in the present paper. However, out of these sixteen specializations, none is relevant to the specializations selected for the present research.

The ACM Computing Classification System [1] is used to classify publications, which is a different purpose than the characterization of specializations. Still, the classification system is relevant enough for the present paper. It lists eleven main categories called first-level nodes. Out of these main categories, at least (a) C. Computer Systems Organization is relevant to Computer Systems specializations, (b) E. Data and F. Theory of Computation are relevant to Theory specializations, and (c) I. Computing Methodologies is relevant to Artificial Intelligence specializations.

3. RESEARCH METHOD

The used research method was content analysis, sometimes called document analysis. The basic goal of content analysis "is to take a verbal, non-quantitative document and transform it into quantitative data (Bailey, 1978)" [3, p. 164].

Degree requirements were used as the data source instead of, for example, job advertisements. We have conducted a job advertisement analysis previously in the area of software systems [15]. Based on our experience, this would probably not work for purely academic specializations such as Artificial Intelligence and Theory for several reasons; for example, it would be hard to find enough suitable advertisements. Job advertisements would be a good data source in the area of computer systems but for the sake of consistency, the same data source was used for all three specializations.

3.1 Sampling

American data sources were used for practical reasons and in order to get results that would be probably interesting for a wider readership. For graduate programs, the 32 best institutions were selected from U.S. News [18] ranking list for computer science graduate programs. For undergraduate programs, the 59 best institutions were selected from the same list. The sample was greater for the undergraduate programs because they offered specializations less often than the graduate programs did.

From each institution, the degree requirements were sought from the web pages of the institution. The data were gathered in July and August 2006, and most of the data were from the academic years 2005–2006 and 2006–2007.

Bachelor's of Science in Computer Science was selected if an institution offered several undergraduate programs in computing. The closest alternative was selected if no such program was offered.

Master's of Science in Computer Science was selected if an institution offered several graduate programs in computing. The closest alternative was selected if no such program was offered. The Doctoral program was selected if no Master's program was offered or the Master's program did not offer specializations but the Doctoral program did.

3.2 Coding

First, the coding of the specializations is explained. Then, the course coding is presented.

3.2.1 Specializations

Typically, in an American computer science program at a research university, a student may or has to choose one specialization out of 3–10 alternatives. Next are explained how the specializations were classified. For example, it had to be decided which specializations were classified to the category Artificial Intelligence.

In some cases, other areas than specializations were used if specializations were not offered. Such areas were typically called breadth or diversity requirements and the number of areas was small, from two to five. For example, a student had to take at least one course from each of three breadth areas and a breadth area listed 3–5 courses. We considered using breadth areas instead of specializations as a small problem because data from breadth areas showed topics that were considered central to that area. This decision was made because specializations were rare in undergraduate programs.

Classification of the category Artificial Intelligence was straightforward because most suitable specializations were named as Artificial Intelligence. Intelligent Systems specializations were included as well.

The categories Computer Systems and Theory were more difficult to classify than Artificial Intelligence. A Computer Systems specialization was always included in the category Computer Systems. A Systems specialization was included if both software and hardware courses were required or elective. A Systems specialization was classified as Software Systems if it was purely software-oriented. Computer Architecture and Hardware specializations were always classified into the category Hardware, not into Computer Systems.

Specializations Algorithms, Formal, Theoretical Computer Science, Theory, and Theory of Computation were always included in the category Theory. A specialization Foundations of Computer Science was included if it was suitable enough according to the characterization presented in Section 2.

Altogether, the selected degree programs offered 319 specializations of which 93 (29%) course names were read before classification. In 71% of the cases, the classification was based on a specialization name only. More details of the classification are presented in Appendix A.

Proportion was counted for each specialization. Greater proportion means that a specialization was offered more often. For example, 24 undergraduate programs offered specializations and 16 of these programs offered a specialization in Computer Systems. Thus, the proportion of Computer Systems was 67% for the undergraduate programs.

3.2.2 Courses

Typically, in an American computer science program at a research university, a specialization consists of 3–5 courses that are required or elective. Next are explained how the courses were classified. For example, it had to be decided which courses were classified to the category Operating Systems.

Data was analyzed twice. The first round was more superficial because only course names were used. During the second round, the course descriptions were read as well and the courses were classified using predefined course definitions (Appendix B). We wrote Appendix B using mainly the results of the first round, CC2001 [5], and the course catalogs of our institution [8; 9]. During the second round, a course was classified into the category "Not found" if we did not find its description from the web. Altogether, the six subsamples included 528 courses of which 23 (4%) descriptions were not found. A course was classified into the category Other if the course description was found but we were not able to classify the course using Appendix B. One hundred and seventy (32%) courses were classified into the category Other.

For the courses, weighted averages were counted instead of proportions. Counting weighted averages was more complex but they were used in order to take account whether a course was required or elective. Typically, a specialization included a set of courses that were required or elective. For example, a student had to take two required courses and choose one course from the list of five. One could assume that required courses were more central than elective courses for a given specialization. Each course was given weight 1 if the course was required. The weight was counted according to the number of elective courses if the course was elective. For example, the weigh was 0.2 when a student had to choose one course out of five.

Finally, weighted averages from 0 to 1 were counted for each course category. For example, the weighted average would be 0.46 [(4 * 1 + 3 * 1/5)/10 = 0.46] if the number of specializations was ten, the course was required in four specializations, and offered as elective (choose one out of five) in three specialization. Therefore, greater weighted average means that a course was more common or central to the given specialization. The maximum average 1 would mean that the course was required in every specialization.

3.3 Changes Compared with Previous Research

Compared with our previous research [16], the following changes were made into the research methodology: (a) The sample size of institutions was considerably greater for the undergraduate (N = 59) than for the graduate programs (N = 32) because specializations were offered less often in the undergraduate programs. The goal was that the size of each subsample for a specialization should be at least ten. (b) Only U.S. News [18] ranking list was used because it was the most recent. Previously also the ranking list by Geist and others [6] was used. However, Geist and others' list is already ten years old. (c) The descriptions of all courses were read before a course was classified. Previously, less than 10% of the course descriptions was read and therefore, the classification was based mostly on the course names. (d) For courses, weighted averages were counted instead of proportions. This should show more accurately which courses were required and thus considered to be most central for a certain specialization. Previously required and elective courses were weighted equally. (e) The details of the classification are presented in the appendices in order to make it clearer what specialization and course categories stand for. Previously, no such documentation was provided. (f) Course prerequisites were not analyzed.

4. RESULTS

First, information on the selected institutions is presented. Then, the most common specializations in the degree programs are presented. Finally, the most common courses of the selected three specializations are presented.

4.1 Selected Institutions

According to the Carnegie Classification of Institutions of Higher Education [2], all selected institutions belonged to the category "Research Universities (very high research activity)." Thus, the samples were not representative relative to all institutions that offered computing programs.

4.2 Specializations

The most common specializations in the selected degree programs are presented in Table 1. A specialization is shown in the table if its proportion was at least 10% in the undergraduate or graduate programs. The rows are ordered first according to the column Undergraduate and then according to the column Graduate.

As expected, the graduate programs offered specializations more often than the undergraduate programs did. Ninety-one percent of the graduate programs offered specializations or used equivalent classifications when the proportion was 41% for the undergraduate programs. These are the subsamples of Table 1 (n = 24 and n = 29, respectively).

The proportion of Theory is greater than 100% for the graduate programs because some programs offered more than one specialization that were classified into this category. For example, two specializations were counted if a program offered the specializations "Algorithms" and "Theory of Computation," more information is presented in Appendix A. Similarly, the proportions of the category Other are greater than 100%.

A similar table was presented in our previous paper [16]. There are some differences but for brevity, they are not explained here. For the purposes of the present paper, it is enough to notice that the specializations Artificial Intelligence, Computer Systems, and Theory are still common.

Table 1. Proportions (%) of offered specializations in selected degree programs

Specialization	**Under-graduate** (n = 24)	**Graduate** (n = 29)
Theory	75	110
Computer Systems	58	41
Artificial Intelligence	42	69
Hardware	42	28
Software Systems	33	34
Computer Graphics	29	45
Programming Languages	29	41
Scientific Computing	29	38
Computer Networks	25	21
Applications	25	10
Databases	17	34
Usability	13	7
Software Engineering	8	17
Other	146	131

4.3 Courses

The most central courses of the specializations in Artificial Intelligence, Computer Systems, and Theory are presented in Sections 4.3.1, 4.3.2, and 4.3.3. A course is presented if its weighted average is at least 0.02. However, a course is shown even if its average is smaller than 0.02 when it belongs to the used classification category. The courses are ordered first according to the average and then according to the name. The category Not found/Other is presented last and was explained in Section 3.2.

4.3.1 Artificial Intelligence

The most common courses of the specializations in Artificial Intelligence are presented in Table 2 and Table 3. One could expect that an introductory course Artificial Intelligent was required in every undergraduate specialization and therefore, its weighted average in Table 2 should be 1. However, this was not the case. The weighted average was 0.58 because a course Artificial Intelligence was required in three and elective in six undergraduate specializations. For example, at the undergraduate program of Brown University, a student had to choose one out of four courses: CS141 Introduction to Artificial Intelligence, CS143 Introduction to Computer Vision, CS148 Building Intelligent Robots, or CS149 Introduction To Combinatorial Optimization.

Table 2. Most common courses (n = 49) **of Artificial Intelligence specializations in undergraduate programs** (n = 9)

Course	Weighted average
Artificial Intelligence	0.58
Robotics	0.25
Computer Vision	0.24
Natural Language Processing	0.14
Machine Learning	0.12
Neural Networks	0.11
Advanced Artificial Intelligence	0.10
Expert Systems	0.01
Multi-Agent Systems	0.00
Planning and Reasoning Systems	0.00
Other/Not found	0.57

Table 3. Most common courses (n = 107) **of Artificial Intelligence specializations in graduate programs** (n = 16)

Course	Weighted average
Artificial Intelligence	0.27
Advanced Artificial Intelligence	0.24
Machine Learning	0.19
Natural Language Processing	0.14
Planning and Reasoning Systems	0.10
Computer Vision	0.06
Robotics	0.06
Expert Systems	0.05
Multi-Agent Systems	0.03
Neural Networks	0.02
Other/Not found	0.53

The results are compared with CC2001 [5] and the ACM Computing Classification System [1] because they are most relevant. According to CC2001 (p. 235), the following courses belong to the area Intelligent Systems: Intelligent Systems, Automated Reasoning, Knowledge-Based Systems, Machine Learning, Planning Systems, Natural Language Processing, Agents, Robotics, Symbolic Computation, and Genetic Algorithms. The CC2001 list matches reasonable well with our results. The biggest differences are that (a) Symbolic Computation and Genetic Algorithms are not central according to our results and (b) Computer Vision is central according to our results but it was not in the CC2001 list.

According to the ACM Computing Classification System [1], the sublevels of the second-level category I.2 Artificial Intelligence are as follows: I.2.0 General, I.2.1 Applications and Expert Systems, I.2.2 Automatic Programming, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods, I.2.5 Programming Languages and Software, I.2.6 Learning, I.2.7 Natural Language Processing, I.2.8 Problem Solving, Control Methods, and Search, I.2.9 Robotics, I.2.10 Vision and Scene Understanding, I.2.11 Distributed Artificial Intelligence, and I.2.m Miscellaneous. These categories match reasonable well with the top parts of Table 2 and Table 3. The biggest differences are that Automatic Programming and "Programming Languages and Software" are not central according to our results. There are also other categories of the classification system that are not apparent in Table 2 and Table 3 but these are typical subtopics for an Introduction to Artificial Intelligence course (e.g, I.2.8 Problem Solving, Control Methods, and Search).

4.3.2 Computer Systems

The most common courses of the specializations in Computer Systems are presented in Table 4 and Table 5. The results are compared with CC2001 [5], CE2004 [14], and the ACM Computing Classification System [1] because they are most relevant. According to CC2001 (p. 235), the following courses belong to the areas "Architecture and Organization" and Operating Systems: Advanced Computer Architecture, Parallel Architectures, System on a Chip, VLSI Development, Device Development, Advanced Operating Systems, Concurrent and Distributed Systems, Dependable Computing, Fault Tolerance, Real-Time Systems. The CC2001 list matches reasonable well with Table 4. The biggest differences are that (a) Parallel Architectures, Concurrent and Distributed Systems, Dependable Computing, and Fault Tolerance are not central according to our results. The CC2001 course names System on a Chip and Device Development probably refer to the same type of courses as our category Design of Digital Systems.

Thirty-three courses are presented in the CE2004 list of advanced courses [14, p. 35]. Out of these courses, only Advanced Computer Architecture matches with the results of Table 4. This is not surprising because in a computer engineering program related topics are partly covered in introductory and intermediate courses. For example, the courses Computer Architecture and Computer Networks are scheduled as required third year courses in a model program [14, B.5], not as elective courses. It is not reasonable to compare whether the CE2004 list contains courses that are not mentioned in Table 4 because the CE2004 list is not divided into areas.

According to the ACM Computing Classification System [1], there are seven sublevels of the first-level category C. Computer Systems Organization: C.0 General, C.1 Processor architectures, C.2 Computer-communication networks, C.3 Special-purpose and application-based systems, C.4 Performance of systems, C.5 Computer system implementation, and C.m Miscellaneous. The categories C.1 and C.2 match with our results but the other categories do not match or are too general to compare. To sum up, the match is only satisfactory.

Table 4. Most common courses (*n* = 75) of Computer Systems specializations in undergraduate programs (*n* =12)

Course	Weighted average
Operating Systems	0.41
Computer Networks	0.26
Computer Architecture	0.24
Design of Digital Systems	0.20
Advanced Computer Architecture	0.15
Digital Logic	0.13
Distributed Systems	0.09
Compilers	0.08
Databases	0.06
Embedded Systems	0.04
Programming Languages	0.03
Other	0.30

Table 5. Most common courses (*n* = 112) of Computer Systems specializations in graduate programs (*n* = 13)

Course	Weighted average
Advanced Operating Systems	0.17
Computer Architecture	0.13
Advanced Computer Architecture	0.12
Advanced Computer Networks	0.12
Programming Languages	0.11
Design of VLSI Circuits	0.10
Computer Networks	0.09
Distributed Systems	0.07
Operating Systems	0.06
Advanced Compilers	0.05
Computer Security	0.05
Compilers	0.04
Databases	0.04
Design of Digital Systems	0.04
Embedded Systems	0.03
Advanced Databases	0.02
Digital Logic	0.01
Other	0.37

4.3.3 Theory

The most common courses of the specializations in Theory are presented in Table 6 and Table 7. One could expect that the weighted averages of some courses would be greater, in particular the average of Data Structures and Algorithms. However, this course was often not mentioned at all in the requirements of a specialization. The obvious explanation is that in some degree programs Data Structures and Algorithms is required for every student and it is studied already during the first, second, or third year.

The results are compared with CC2001 [5] and the ACM Computing Classification System [1] because they are most relevant. According to CC2001 (p. 235), the following courses belong to the areas Discrete Structures and "Algorithms and Complexity:" Combinatorics, Probability and Statistics, Coding and Information Theory, Advanced Algorithm Analysis, Automata and Language Theory, Cryptography, Geometric Algorithms, and Parallel Algorithms. The courses "Probability and Statistics" and "Coding and Information Theory" of the CC2001 list are not mentioned in Table 7. In addition, Table 7 contains several courses such as Machine Learning that are not mentioned in the CC2001 list. Anyhow, the overall match is reasonably good.

According to the ACM Computing Classification System [1], there are thirteen sublevels of the first-level categories E. Data and F. Theory of Computation: E.0 General, E.1 Data structures, E.2 Data storage representations, E.3 Data encryption, E.4 Coding and information theory, E.5 Files, E.m Miscellaneous, F.0 General, F.1 Computation by abstract devices, F.2 Analysis of algorithms and problem complexity, F.3 Logics and meanings of programs, F.4 Mathematical logic and formal languages, and F.m Miscellaneous. The category F. Theory of Computation matches well with our results but the category E. Data only moderately.

Table 6. Most common courses (*n* = 116) of Theory specializations in undergraduate programs (*n* = 16)

Course	Weighted average
Design and Analysis of Algorithms	0.36
Computational Complexity	0.21
Theory of Computation	0.20
Advanced Data Structures and Algorithms	0.13
Graph Theory	0.11
Cryptography	0.10
Data Structures and Algorithms	0.09
Logic	0.09
Parallel Computation	0.07
Programming Languages	0.07
Formal Languages and Automata	0.04
Combinatorial Optimization	0.03
Computational Geometry	0.02
Machine Learning	0.02
Verification	0.01
Other/Not found	0.93

Table 7. Most common courses (*n* = 69) of Theory specializations in graduate programs (*n* = 11*)

Course	Weighted average
Design and Analysis of Algorithms	0.33
Advanced Data Structures and Algorithms	0.22
Theory of Computation	0.16
Logic	0.15
Computational Complexity	0.13
Cryptography	0.13
Computational Geometry	0.03
Parallel Computation	0.03
Combinatorial Optimization	0.02
Data Structures and Algorithms	0.02
Machine Learning	0.02
Formal Languages and Automata	0.00
Graph Theory	0.00
Programming Languages	0.00
Verification	0.00
Other/Not found	0.78

*) The original subsample was 22 graduate programs but it was cut in half in order to reduce classification work.

5. DISCUSSION

The results of the present research are original in a sense that similar results have not been published previously. However, the results are not surprising because they generally match well enough with the related recommendations or classifications.

The results can be classified as conservative as a consequence of the selected research methodology. A different target group could have been selected if the purpose was to find alternatives that were less conservative or even dramatically different. For example, software engineering or computer science programs in Brazil, Russia, India, and China—known also as BRIC countries—might be a more suitable target group in that case. However, this was not the purpose of the present research. Instead, the purpose was to solve common requirements in the area of advanced computer science studies that the current ACM curricula recommendations cover poorly. The ACM curricula recommendations can be classified as normative studies. Our research rather supports or complements these normative studies than challenges them.

The results of most common courses (Section 4.3) should be quite similar regardless the country because they show relationships between the various subject matters. What can differ strongly from a country to another, is how commonly a certain specialization is offered; that is, the results of Section 4.2. Indeed, it is likely that Computer Systems specializations are less commonly offered outside the USA. A possible explanation is that American companies such as IBM and Hewlett-Packard are major designers and manufactures of computers. The situation is different in many other countries where computer engineering industry is not as strong. For example in Finland, Computer Systems specializations are rarely offered in the universities. However, Computer Systems specialization is not targeted to computer engineering positions only but might be useful for systems administration positions as well. According to the draft of Computing Curricula 2005 [12, p. 31]: "…, there is a fourth career path that CS programs do not target but nonetheless draws many computer science graduates: *Career Path 4: Planning and managing organizational technology infrastructure*."

As far as we know, the same limitation is not true for Artificial Intelligence and Theory; that is, these specializations are offered quite often outside the USA as well. These two specializations are interesting because they can be characterized as traditional academic specializations and as being more academic than many other specializations. Here, more academic means that a specialization is not at all or only rarely offered at community colleges; that is, it is specific to universities. Some other specializations are less academic in that sense that they are offered at the community colleges as well. For example, specializations Software Engineering, Databases, and Computer Networks are often or sometimes offered at Finnish community colleges.

Specializations Computer Systems and Theory were much more difficult to analyze than Artificial Intelligence for several reasons. Apparently, these two concepts refer to broader areas than specializations on average.

6. ACKNOWLEDGMENTS

We thank Prof. M. Syrjänen, Dr. T. Janhunen, Dr. V. Hirvisalo, and H. Arppe from the Helsinki University of Technology for commenting on the manuscripts of the present paper.

7. REFERENCES

[1] Association for Computing Machinery. *The ACM Computing Classification System [1998 Version]*. Retrieved on August 11, 2006, from ACM web site: http://www.acm.org/class/1998/. 1998.

[2] Carnegie Foundation. *The Carnegie Classification of Institutions of Higher Education*. Retrieved on August 15, 2006, from Carnegie Foundation web site: http://www.carnegiefoundation.org/classifications/.

[3] Cohen, L., Manion, L., and Morrison, K. *Research Methods in Education* (5th ed.). RoutledgeFarmer, London, 2000.

[4] Educational Resources Information Center. *ERIC Thesaurus*. Retrieved October 31, 2005, from Educational Resources Information Center web site: http://www.ericfacility.net/extra/pub/thessearch.cfm.

[5] Engel, G., and Roberts, E. (Eds.). *Computing Curricula 2001: Computer Science*. IEEE Computer Society and Association for Computing Machinery. Retrieved on October 18, 2002, from IEEE Computing Society web site: http://www.computer.org/education/cc2001/final/cc2001.pdf.

[6] Geist, R., Chetuparambil, M., Hedetniemi, S., and Turner, A. J. Computing research programs in the U.S. *Communications of the ACM, 39*, 12 (Dec. 1996), 96-99.

[7] Gorgone, J.T, and Gray, P. (Eds.). MSIS 2000: Model Curriculum and Guidelines for Graduate Degree Programs in Information Systems. *Communications of the Association for Information Systems*, vol. 3, January 2000. Retrieved on August 12, 2006, from ACM web site: http://www.acm.org/education/curricula.html#MSIS2000.

[8] Helsinki University of Technology. *Study programme. ECTS guide 2003–2004*. 2003.

[9] Helsinki University of Technology. *Study programme. ECTS guide 2006–2007*. 2006.

[10] International Organization for Standardization. *Information technology. Vocabulary. Part 1: Fundamental terms*. 3rd ed. ISO/IEC 2382–1: 1993 (E/F). 1993.

[11] Radatz, J. (Ed.). *The IEEE standard dictionary of electrical and electronics terms*. Institute of Electrical and Electronics Engineers. 1996.

[12] Shackelford, R., Cross, J. H., II, Davies, G., Impagliazzo, J., Kamali, R., LeBlanc, R., et al. (2005). *Computing Curricula 2005. The overview report*. Draft, April 11, 2005. Retrieved on June 21, 2005, from ACM web site: http://www.acm.org/education/Draft_5-23-051.pdf.

[13] Shapiro, S.C. (Ed.). *Encyclopedia of Artificial Intelligence*, 2nd ed. Wiley, New York.

[14] Soldan, D., Hughes, J.L.A, Impagliazzo, J., McGettrick, A., Nelson, V.P., Srimani, P.K., Theys, M.D. *Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*. Retrieved on August 12, 2006, from ACM web site: http://www.acm.org/education/curricula.html#CE2004. 2004.

[15] Surakka, S. Analysis of technical skills in job advertisements targeted at software developers. *Informatics in Education, 4*, 1, 101-122. 2005.

[16] Surakka, S. Specialization in Software Systems: Content Analysis of Degree Requirements. In T. Salokoski, T. Mäntylä, and M. Laakso (Eds.). *Proceedings of Koli*

Calling. Fifth Koli Calling Conference on Computer Science Education, November 17-20, 2005, Koli, Finland. TUCS General Publication, 41, pp. 162-165. URL: http://www.it.utu.fi/koli05/proceedings/final_composition.b5.060207.pdf.

[17] *Theoretical Computer Science*. Retrieved August 28, 2006, from Elsevier web site: http://www.elsevier.com/wps/find/journaldescription.cws_home/505625/description#description.

[18] U.S. News & World Report. *America's Best Graduate Schools 2007*. Premium online edition. 2006. (URL is not provided because this is a charged service.)

APPENDICIES

Appendix A: Classification of Specializations

Table A1 presents how the specializations were classified to the categories used in the present paper. Not all original names of the data set are presented in the table but only typical ones. For example, the specialization Intelligent Systems was classified into category Artificial Intelligence.

Table A1. Categories of specializations

Category	Specialization name
Applications	Applications
Artificial Intelligence	Artificial Intelligence Intelligent Systems
Computer Systems	Computer Systems Operating Systems* Systems*
Computer Graphics	Computer Graphics Graphics
Computer Networks	Computer Networks Networking and Communications
Databases	Database Systems Databases
Hardware	Computer Architecture Hardware
Other	A specialization that was not classified to the other categories
Programming Languages	Compilers Programming Languages
Scientific Computing	Numerical Analysis Scientific Computing
Software Engineering	Software Engineering
Software Systems	Operating Systems* Software Software Systems Systems*
Theory	Algorithms Formal Foundations of Computer Science* Theoretical Computer Science Theory Theory of Computation
Usability	Human-Computer Interaction Usability

* An asterisk means that a specialization was classified to one of alternative categories according to the course requirements. For example, a Systems specialization was classified to the category Computer Systems or Software Systems according to the course requirements while a Theory specialization was always classified to the category Theory.

Appendix B: Course Definitions

The definitions used to classify the courses are presented next. Artificial Intelligence courses are presented in Appendix B.1, Computer Systems courses in B.2, and Theoretical Computer System courses in B.3. Inside each list, the courses are ordered by name.

A CC2001 course description was used when available. In most cases, the course descriptions of our institution [8; 9] were used. A description from some other institution was used when no suitable course was offered at our institution. These references to the web pages are presented only here, not in the reference list.

No course by course definitions are presented for the courses of which names include the word "Advanced." For example, an Advanced Artificial Intelligence course covers the same type of topics as Introduction to Artificial Intelligence but at a more advanced level. An alternative course name would be Artificial Intelligence II when Artificial Intelligence I is used for an introductory course. Typically, an introductory course is a prerequisite for an advanced course.

B.1 Artificial Intelligence

Artificial Intelligence
: Introduction to artificial intelligence. See the CC2001 definition of CS260 Artificial Intelligence [5, p. 220].

Computer Vision
: Introduction to the use of computers in analysis of visual data. Image forming, image geometry, low-level vision, motion perception, feature extraction, 2D area, and 3D object representation, forming and recognition of structural patterns. [9, p. 96]

Expert Systems
: Design principles of computer system designed to solve a specific problem or class of problems by processing information specific to the problem domain. A domain area can be, for example, law or medicine. [13, p. 380]

Machine Learning
: Machine learning studies the automated acquisition of expert knowledge. Representation of experience and acquired knowledge. Defining the performance task. Supervised and unsupervised learning. Incremental and nonincremental learning. Inductive and analytic learning. [13, pp. 785–788]

Multi-Agent Systems
: Theory, architectures, and applications for agent-based computing. Decision-making on the basis of uncertain information. [8, p. 83]

Natural Language Processing
: Overview of application of statistical and adaptive methods for analysis of natural language, for example, the analysis, organization and search from text collections, language modeling for natural language recognition, syntactic and semantic analysis, probabilistic grammars and parsing, and statistical machine translation. [9, p. 95]

Neural Networks
: Introduction to neural networks, learning processes, single layer networks, multi-layer perception networks and back-propagation algorithm, radial-basis function networks, self-organizing maps and learning vector quantization. [9, p. 95]

Planning and Reasoning Systems
: Planning can be thought of as determining all the small tasks that must be carried out in order to accomplish a goal. Constraints and scheduling. Different types of reasoning such as case-based, causal, and commonsense

reasoning. [13, pp. 1159 and 1265–1339, and http://ic.arc.nasa.gov/projects/remote-agent/pstext.html].

Robotics
: Basics in robotics. Industrial robots and mobile robots. Subsystems and physical component of robots. Basic kinematics and motion control principles. Examples of various practical applications of robotics. [9, p. 46]

B.2 Computer Systems

Compilers
: Introduction to compilers. See the CC2001 definition of CS240 Programming Language Translation [5, p. 215].

Computer Architecture
: Introduction to computer architecture. See the CC2001 definition of CS220 Computer Architecture [5, p. 206].

Computer Networks
: Implementation principles of telecommunications software and fundamental principles of computer networks, especially IP networks. Routing, name service, network administration, protocol development and network programming. [9, p. 110]

Computer Security
: Basic methods for implementing security and applying them. Building safe systems. Identification, authentication and access control. Possibilities offered by cryptography. Security models. Security of operating systems and services. [9, p. 110]

Databases
: Introduction to database systems. See the CC2001 definition of CS270 Databases [5, p. 224].

Digital Logic
: Introduction to digital systems. Design and implementation methods of digital electronic devices. Coupling functions, combinatorial and sequential logic, MSI and LSI circuits, PLA circuits, memory, timing and interface issues, digital arithmetic and codes. [9, p. 139]

Design of Digital Systems
: Design and laboratory exercises concerning digital circuits and devices and electrical phenomena. [9, p. 140]

Design of VLSI Circuits
: Implementation of digital logic elements: combinational circuits, latches and flip-flops. Synchronous and asynchronous digital systems. Implementation of finite state machines, testing of digital circuits. [9, p. 139]

Distributed Systems
: Architecture and implementation techniques of distributed systems. [9, p. 108]

Embedded Systems
: Development of embedded systems, their hardware, solutions and application areas. Programming with real-time, fault-tolerance, and correctness requirements. [9, p. 108]

Operating Systems
: Introduction to operating systems. See the CC2001 definition of CS225 Operating Systems [5, p. 210].

Programming Languages
: The interpretation of the typical mechanism of programming languages. The concepts, structure, and implementation of interpreters. Definition and implementation of domain-specific languages. [9, p. 105]

B.3 Theory

Combinatorial Optimization
: Algorithms for matching problems. Introduction to matroids; polyhedral combinatorics. Complexity theory and NP completeness. Perfect graphs and the ellipsoid method. [http://www.college.columbia.edu/…]

Computational Complexity
: NP-completeness. Randomized algorithms. Cryptography. Approximation algorithms. Parallel algorithms. Polynomial hierarchy. PSPACE-completeness. [9, p. 102]

Computational Geometry
: Introductory course to computational geometry. Designing and analyzing efficient algorithms and data structures for computational problems in discrete geometry, such as convex hulls, geometric intersections, geometric structures such as Voronoi diagrams and Delaunay triangulations, arrangements of lines and hyperplanes, and range searching. [http://www.cs.umd.edu/class/fall2005/cmsc754/]

Cryptography
: Data and communications security. Principles of cryptographic security. Symmetric cryptosystems. Stream ciphers. Block ciphers: DES, IDEA, AES. Modes of operation. Asymmetric cryptosystems. Digital signatures. Authentication and key agreement. Applications: SSL, TLS, IPSec, GSM, Bluetooth. [9, p. 101]

Data Structures and Algorithms
: Introduction to data structures and algorithms. See the CC2001 definition of CS103I Data Structures and Algorithms [5, p. 165].

Design and Analysis of Algorithms
: Introduction to design and analysis of algorithms. See the CC2001 definition of CS210 Algorithm Design and Analysis [5, p. 204].

Formal Languages and Automata
: An introduction to the fundamental ideas and models underlying computing: finite automata, regular sets, pushdown automata, context-free grammars, Turing machines, undecidability, and complexity theory. [http://www.cs.cmu.edu/afs/cs/usr/cathyf/www/ugcourses.htm]

Graph Theory
: Introduction to graph theory. Trees, planar graphs and digraphs. Graph coloring. Random graphs. Algorithms for central graph problems. Applications. [9, p. 102]

Logic
: Introduction to logic. Propositional and predicate logic, their syntax, semantics and proof theory. Applications of logic in computer science. [9, p. 101]

Machine Learning
: See Appendix B.1.

Parallel Computation
: Fundamental theoretical issues in designing parallel algorithms and architectures. Shared memory models of parallel computation. Parallel algorithms for linear algebra, sorting, Fourier Transform, recurrence evaluation, and graph problems. Interconnection network based models. Algorithm design techniques for networks like hypercubes, shuffle-exchanges, threes, meshes and butterfly networks. Systolic arrays and techniques for generating them. Message routing. [http://www.eecs.berkeley.edu/Gradnotes/Content/Section6.pdf]

Programming Languages
: See Appendix B.2.

Theory of Computation
: Introduction to theory of computation. Finite automata and regular languages. Context-free grammars and pushdown automata. Context-sensitive and unrestricted grammars. Turing machines and computability. [9, p. 101]

Verification
: Verification and analysis of parallel and distributed systems using computer aided tools. Practical verification methods, e.g. partial reachability analysis. [8, p. 83]

Program Working Storage: A Beginner's Model

Evgenia Vagianou

Computer Information Systems Department
The American College of Greece
6 Gravias str., Ag. Paraskevi
15342, Athens, Greece
jes@acgmail.gr

Department of Informatics
University of Sussex
Falmer, Brighton
BN1 9QH, UK
ev38@sussex.ac.uk

ABSTRACT

The aim of this paper is to introduce and validate the concept of *program working storage* (PWS) as a) a means of smooth transition of students in introductory programming courses from the *end-user stance* to the *programmer stance*, and b) a system which can provide comprehensive understanding of certain difficult programming concepts. In this respect, the program-memory interaction is considered as a possible "threshold concept" [31, 33]. Based on constructivism [16, 23, 41, 42], the PWS is then discussed as a potential beginner's viable model, which can be, later on, *refined* to what Ben-Ari describes as a viable computer model [5]. The extent to which the PWS can be used as a conceptual framework, which will enable teachers and learners to focus on program-memory interaction across a variety of dimensions, and eventually relate them to form a coherent whole, is also examined. The exact implementation of the PWS in the context of the various programming languages is beyond the scope of this paper. Nevertheless, it constitutes a topic for detailed study and future research.

Keywords

Teaching programming, introductory programming, constructivist instruction, preconceptions, threshold concepts.

1. INTRODUCTION

Computing curricula, in undergraduate institutions, are usually shaped with respect to the computing disciplines offered (i.e. CE, CS, IS, IT, SE), and the course targeting approach[1]. Due to cost-related issues, broad student audience strategies, and the fact that the strongest commonality across all disciplines is "concepts and skills of computer programming", a *programming-first* model is usually adopted [1,2]. Inevitably, introductory programming courses often serve as the actual starting point of study.

Empirical studies, relative to novice programmers and introductory programming courses, yield that students find the learning of programming particularly difficult [15, 27, 44]. Their prior knowledge and understanding of the domain (or related domains) affect the manner in which they engage learning [33]. An interesting, though expected, observation is the common perspective of students in introductory programming courses and end-users, who regard a program as the "magical instrument" that "will do the job". Frequently expressed with comments like "it didn't work" or "it did it again", the above perspective also implies the notion of a "someone" or "something" else being responsible for what is happening.

It seems safe to claim that in the mind of the end-user, "the program" (or "the computer") constitutes a distinct but irrelevant to his/her concerns ontology, since it only serves as a tool, while the point of interest is the task to be accomplished. Such conceptions can be justified as viable in the sense that they are consistent with the context in which they were created [40, 42]. Nevertheless, when novice programmers carry them in a new context, they can prove inefficient [36, 40] and even haphazard [4]. According to the constructivist theory of learning, though, preconceptions are crucial in that they form the basis for new knowledge to be built.

Educators of introductory programming courses are called to deprecate the *end-user stance* of their students and enforce the *programmer stance*, which must further comply with a more consistent *computing expert stance*[2]. This has to be achieved in a specified time period, which compared to the potentially vast amount of new information, concepts and perspectives that a student needs to perceive and interpret, is so inadequate that selection of focus is inevitable. The practical constrains of teaching objectives and overlapping course material are just the final "straws" on the pile.

The aim of this study is to introduce and validate the concept of *program working storage* (PWS) as a) a means of smooth transition of students in introductory programming courses from the *end-user stance* to the *programmer stance*, and b) a system which can provide comprehensive understanding of certain difficult programming concepts. In this respect, the program-memory interaction is examined as a possible "threshold concept" [31, 33], and the PWS is discussed as a potential beginner's viable model, which can be, later on, *refined* to a viable computer model [5]. The suggested approach, although constructivistic in nature, has a behaviouristic dimension, since one of its goals is that the PWS is eventually automatically recalled.

The PWS is also discussed from the perspective of a conceptual framework, which will enable teachers and learners to focus on program-memory interaction across a variety of dimensions (such as data types, implicit/explicit reference, states, etc.) and eventually relate them to form a coherent whole.

The suggested representation of a PWS is graphical with limited constrains to its precise deployed form.

[1] According to the CC2005 Computing Curricula Task Force Report, there are two possible approaches: the filter, which recommends parallel discipline-specific introductory course sequences, and the funnel, which recommends a common introductory sequence [1].

[2] Terms are discussed in section 2.4.

The exact implementation of the PWS in the context of the various programming languages is not in the scope of this paper and, therefore, not extensively addressed. Nevertheless, it constitutes a topic for detailed study and future research.

2. LITERATURE REVIEW

The literature related to this study encompasses research in a variety of fields, which, due to space limitations, are divided into three basic categories: novice programming issues, teaching and learning theories, mental models and graphical external representations.

2.1 Novice Programming Issues

The research in novice programmers' problems, misunderstandings, bugs etc., is very large and detailed [e.g. 9, 14, 24, 35, 36], and the literature addressing ways to cope with them is analogous [4, 14, 18, 20, 44]. This major concern of educators may be fairly attributed to two (2) reasons:

a) Programming involves specifying behaviour that will occur in the future [6, 7]. It, therefore, imposes the utilization of certain cognitive skills, which in the case of novice programmers are inert if present at all. In their study, Fix et al. [17] identify several characteristics in experts' mental models, which are not observable in novices' representations, thus implying the absence of the related skills.

b) Existing knowledge and experience lead to preconceptions which, as White [43] notes, easily turn to misconceptions. The main problem seems to be what Pea [36] describes as "conversing with a human", leading to language-independent "bugs". The issue is that natural language pragmatics, intuitively used in human interaction, contradicts with the "mechanistic" rules that a formal system interprets instructions.

There exist a really large number of approaches and tools, which have been developed in order to address the above points. In their majority they utilize a technique, known as "program visualization" [30, 39], which involves exposure of memory contents in order to facilitate program comprehension.

2.2 Teaching and Learning Theories

2.2.1 Behaviourism and Cognitivism

Traditional teaching approaches are based on behaviouristic and cognitive theories, the fundamental assumptions of which are in accordance with the objectivistic philosophical paradigm [16, 23]. Objectivism assumes the world and the mind as two distinct ontologies. The world is viewed as a complete, well-structured reality, and the mind as an abstract processing machine [16]. The goal of teaching is to efficiently "map the structure of the world onto the learner's mind" [16, 23], so that it "mirrors reality" [23].

2.2.2 Constructivism

Constructivism is a philosophical paradigm [16, 23, 41] concerned with the nature of knowledge. Without denying the existence of an objective world, knowledge is not assumed to be a part of it. The main argument is that knowledge is constructed by an individual's own experiences, thus forming one's personal *version* (model) of the world ("known world" [41]).

Constructivism, as a learning theory, encourages the building of knowledge on a subject, by changing the nature of the questions a student asks about it [41]. Usually regarding a subject as a complex system consisting of a number of sub-domains (contextual entities), educators have been deeply concerned with the conceptions (i.e. the models) that students carry in the various contexts. A model developed in a certain context, may prove inadequate in a new situation, resulting to unjustifiable inferences.

A considerable amount of research is concerned with ways of dealing with preconceptions. In their analysis of knowledge in transition [40], Smith et al. argue that the existing approaches[3] involve a significant shift from preconceptions to expert models, by dispelling the former and adopting the latter. They note the anti-pedagogical dimension of such practices[4], as well as, the conflict with the basic premise of constructivism – that knowledge is built "recursively". Eventually, they propose "refinement" as an effective approach. In this case, students acknowledge that their existing knowledge is inadequate to explain phenomena, and transform it into more sophisticated forms through relatively stable intermediate states of understanding.

In CS education, much of the research has been performed by Ben-Ari [3, 4]. One of his strongest points, regarding specifically CS education, is that a viable computer model must be present before programming is engaged. This suggestion is supported by prior research [14, 28, 29] and has been defended by recent research [18, 38, 44].

2.2.3 Threshold concepts

There is not much literature, yet, on "threshold concepts" (suggested by Meyer and Land [32, 33]) since it is a quite recent theoretical notion, the roots of which lie in constructivism. Threshold is characterized a difficult core concept of the domain to be studied, which, once understood, provides a broad ground for comprehension of more advanced concepts. Its originators shape the nature of such concepts based on five dominant characteristics [33]:

A threshold concept is "transformative", in the sense that it can provide a significantly improved understanding of the subject-matter.

It is "irreversible", in that, once the perspective it will provide is understood, it is "unlikely to be forgotten".

It is "integrative", in that it exposes the hidden interrelatedness of something.

It may be "bounded", in the sense that the conventional semantics of the language may lead to substantially different inferences.

It may be "troublesome", in that it may involve troublesome knowledge (i.e. "ritual", "inert", "conceptually difficult", "alien", or "tacit" [33]).

The research for CS threshold concepts has only started.

2.3 Mental and External Representations

2.3.1 Mental Models

In the cognitive science literature, there are two (2) main perspectives concerning the nature of mental representations. The theory of "Formal Rules" [11, 12] assumes that humans

[3] "Replacement", "confrontation", "overcoming" [40].

[4] I.e. assessing one's perception as fundamentally wrong.

construct propositional representations, independently of the form of the stimuli that may emerge them. Therefore, the most efficient representational form is considered to be the sentential. In any other case, the objects of the domain will need to be converted to nouns before encoded, which implies extra mental effort.

The alternative theory, "Mental Models" [21, 22], argues that the entities of mental models represent both structure and content and, therefore, may have arbitrary or iconic properties. It is further argued that the structure of a mental model, which is considered its most fundamental property, should be identical to the structure of the spatial relations as they are perceived or conceived.

Based on the above, Boudreau and Pigeau [10] performed an empirical study, in order to test the effectiveness of spatial reasoning using diagrammatic and sentential representations. Their outcomes point that in both cases, humans perform more or less the same spatial reasoning task. Nevertheless, diagrams were significantly favoured in terms of easiness of use and efficiency.

2.3.2 Graphical External Representations

Graphically expressed external representations (ERs) address primarily concept visualization, i.e. "the process of forming a mental image or vision of something not actually present to the sight" [37]. Cognitive scientists have extensively studied the value of developing ERs and their effect on learners' understanding, while their pay-offs, mostly obvious in the special case of diagram use, have been associated with three main information-processing operations [26]: a) correspondence between elements is automatically expressed, b) information needed for the same inference, can be grouped, thus, reducing the amount of search required to locate the information, and c) *perceptual inferences* are supported, which, by utilizing the power of the human visual system, they can replace clumsy serial logical inferences [13].

It is interesting to note that at the level of introductory programming, as argued by Lahtinen and Ahoniemi [25], most of the visualizations concentrate on *presenting* programming concepts, which facilitates mere understanding of the concept rather than appreciation of its application.

2.4 Name Conventions

The terms presented in this section, have not been formally defined. Here follows clarification of their use in this study.

End-user stance: the expression is used to denote the viewpoint of a non-expert towards a computing system, and it primarily addresses the predisposition towards a computer or a program as a distinct, irrelevant to one's concerns ontology.

Programmer stance: the term assumes awareness of one being *directly* involved in the computer operation processes.

Computing-expert stance: the term refers to the *mind-set* of an expert in a specific computing discipline.

Program Working Storage: the expression (also found as "program working memory") has been met in professional development contexts[5]. In general, it refers to the collection of addressable memory areas, while often it can also include implicitly used areas (such as index registers). The exact way, the term is used in this text is explained in detail in section 4.2.

[5] E.g. operating systems and database development.

3. PROGRAM-MEMORY INTERFACING

3.1 The Computer/Program Conception

As already mentioned, the disposition of the end-user is to concentrate on the task to be accomplished, and the "computer" or the "program" is there in order to serve this task. Although in practical terms both conceptions have the same usefulness for the end-user, it is necessary to stress a simple but important difference between them. While the "computer" view addresses a complete computer system, the "program" view assumes the existence of the computer and implies a distinction between them as two discrete entities that interact. Even further, usually as a result of attending a computer fundamentals course, where the basic computer operation is discussed, it seems that there is a rather fair understanding of the fact that programs are placed (loaded) in the main memory in order to be executed.

Whether the computer is using the program or vice versa, is not a question of this study. However, it is worth to note that experts distinguish between the memory area, where the program is stored, and the memory areas that the program *uses*. Under this perspective, given that a program is a programmer's specification, it is the programmer, who, through the program, is using the memory.

3.2 A Possible Threshold

To the author's knowledge, there has been no systematic research on the importance of a good understanding of program/memory interaction. It may be the case that it is so profoundly related to program comprehension and development that, although practiced and used during the course, it is *unconsciously* neglected. Evidence of its significance is additionally provided by the programming environments and the large number of approaches and empirically tested tools – designed either for teaching programming or for program comprehension (e.g. [14, 15, 34]) – which, as part of their functionality, expose the memory contents during program execution.

It seems that program/memory interfacing has a number of attributes that characterize a threshold concept. It can be thought as troublesome, for example, in the sense that while students may understand that a program is using memory, they do not realize *how* such use takes place, or their active role (through the program) in this process ("inert knowledge" [33]).

It can be assumed bounded since the term "memory" in real-life has a considerably different meaning (e.g. refers to both short-term and long-term memory). It may, also, be considered integrative in that it exposes an important part of the "hidden interrelatedness" of hardware with software, the programmer with the computer, etc.

Finally, it is transformative since, once understood, it will significantly shift the mentality of the learner, and, most probably, irreversible; according to Meyer and Land [32, 33], a concept, the acquisition of which leads to an "epiphany", is highly improbable to be forgotten or "unlearned".

4. THE PROGRAM WORKING STORAGE

The preceding discussion suggests that there is sufficient ground to *cultivate* the programmer's mentality, based on

evidently existing understandings of novice programmers. The aim is to cause the development of a mental model, which will be a) viable, so that it facilitates the learning of the programming practice and, therefore, the teaching objectives of an introductory programming course, b) valid, in that it will be consistent with the anticipated computing-expert mentality, and c) potentially *refinable*, in order to form the basis for more complex/specialized future models. To achieve that, the PWS is deployed as a conceptual model, which abstractly, nevertheless accurately, describes this part of the target system in which the programmer is actively involved.

4.1 Explanatory Basis

For better understanding of the ideas presented in the next section, assume a model, to be reflecting programs in general, using a) a *processing specification*, consisting of a set of statements, expressed in a certain notation and accessed in a predetermined manner, and b) a *data storage-repository*, formed by storage areas, accessed selectively as required by a statement.

Every statement expresses an instruction, which has a storage-repository related objective. In this respect, each statement is *accessing* the storage-repository in order to create a new storage area, use the data stored in one or more storage areas, modify the content of a storage area, or *release* a storage area.

Accessing can take place in one of the following ways:

a) Explicitly: the objective of the instruction involves a *specific storage area*, which can, in turn, be accessed in two possible ways:
   - Directly: referencing openly the interested storage area
   - Indirectly: referencing a lead to the interested storage area

b) Implicitly: the objective of the instruction involves *specific data*; the reference to the respective storage area is hidden.

The physical nature, the exact location, and the size of the storage areas, as well as, the implementation of stacks and heaps by the programming languages, are beyond the scope of the presented model. Storage areas, which, for convenience, will also be addressed as "memory areas", are only subject to the way they are accessed.

In the early stages of learning programming, memory constrains (e.g. stack size) and optimization techniques are not really an issue, primarily due to the fact that programs are small, algorithms are rather straightforward, and data structures are elementary.

4.2 The PWS as a Conceptual Model

4.2.1 Definition

In this study, *program working storage* is defined as a dynamic abstract entity, which is *program dependent* and, therefore, meaningful only in the context of a specific program or *program segment*. Program segment refers to a procedure, function, module, or an arbitrary selection of statement-sequence, with respect to the necessary declarations and initializations of the involved data-constructs (e.g. constants, variables, parameters).

The PWS is realized in two dimensions:

a) Space, as the collection of the short-term data storage areas (DASA) that are utilized by the program during execution.

b) Time, in terms of the states of DASAs throughout program execution.

A DASA is defined as an abstract unit, which corresponds to a physical short-term storage area, and has two primary characteristics: a) ability to hold a data value, and b) lifetime.

As data value is regarded any kind of value that can be stored, according to the semantics of the implementing programming language, independently of whether it can be explicitly manipulated. Usually[6], data values include characters, boolean and numeric values, and memory addresses.

As lifetime is regarded the time interval during which a physical storage area that corresponds to a DASA, is reserved by the program. DASAs with the shortest lifetime are usually implicitly accessed and hold literals and return-values. DASAs with the longest lifetime can be explicitly manipulated and represent global variables and constants.

4.2.2 Representation

The nature of the PWS, the students' limited understanding of the memory role, and the fact that the overall process is not evident impose the need of an external representation. Based on the research presented in sections 2.1 and 2.3, a suitable visualization can be justified as the best choice. Figure 1 depicts a simple representation of the PWS at a certain state.

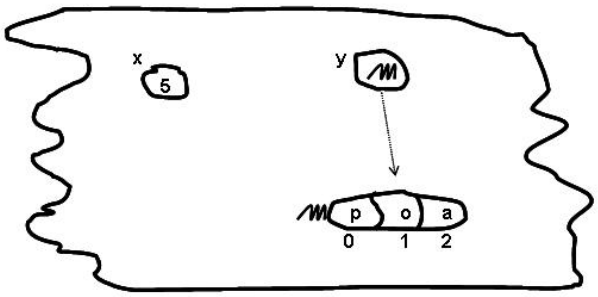

Figure 1

The minimalist form of the representation enables the learners to instantly produce states of a PWS even on simple media like paper, thus enhancing active involvement. On the other hand, teachers may enrich their representations with color, line formats, etc. (what Green and Blackwell have called "secondary notations" [7, 19]) in order to give emphasis or express additional information. Figure 2 depicts an instance of the PWS of the accompanying program. Colour is used to emphasize scope and dashed lines to indicate implicitly accessed DASAs.

[6] Considering the implementation languages used introductory programming courses, which are usually imperative or object-oriented.

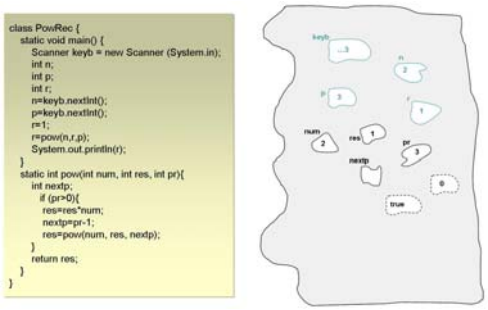

Figure 2

Even though the exact structure of the graphical representation is not a major concern, there are three recommendations:

- The storage-repository (memory at large) is arbitrarily shaped.
- DASAs are placed at relatively random positions.
- DASAs are shaped as to prevent misleading associations[7].

In any case, however, the goal is to draw attention to the significance of the "creation" and use of storage areas, rather than their physical nature.

4.3 The PWS as a Conceptual Framework

The PWS may be employed to present a variety of programming concepts and impinge their interrelatedness. At a basic level of use, it provides a means to identify and discuss memory/storage and data concepts, while parallel assimilation of its two dimensions (space and time) facilitates the study of essential programming constructs and problem-solving techniques.

Studied in space, the PWS can be used to emphasize explicit/implicit reference, as well as, direct/indirect manipulation, and, therefore, reveal major functionality features of variables, constants, parameters, data structures, literals, results of calculated expressions, and return-values. When examined in time, it can be very effective in highlighting scope of data constructs (presence and absence from states), and illustrating challenging attributes of data transition, such as copy and replacement of data-values.

Both dimensions of the PWS can be subject to *depth* in order to serve instruction as required (e.g. focus, student level of understanding). For space, depth refers to levels of abstraction of data structures, which can be suppressed, semi-suppressed, or expanded. For time, abstraction is related to *identifiable* program-execution, relative to memory-use, states. These may include segments, individual statements, or even expressions.

The study of the PWS in parts can also prove useful. Thorough examination of the states of selected DASAs may be used in order to underline the mechanics of control structures (e.g. iteration) and the roles of variables [24]. Selectively focused states may be used to stress the necessity of actions that will trigger the occurrence of the state in question, thus provoking program control flow and sequence. The acknowledgement of the concrete states of a DASA may facilitate the appreciation of the imposing action (statement or program segment), and, therefore contribute to the comprehension of the deployed algorithm.

It is worth to note that elementary optimization issues may also be addressed in terms of the DASAs' frequency of use.

5. DISCUSSION

The effectiveness of revealing the data-flow in accordance with control-flow becomes evident by the variety of approaches that utilize it. In their majority, tools that have been developed to assist the learning of programming include such a component. The PWS, however, attempts to take common practice one step further.

The PWS has been used in introductory programming courses for several years, with two different implementation languages – Pascal and Java. It occurred from the need to deal with novices' preconceptions – such as the conventional notion of memory and the mathematical view of variables – quickly and effectively.

Typical observations regarding the use of the PWS, which outline regular behaviour of approximately 300 students in 16 groups, include the following:

Using the graphical representation, students justify DASAs' requirements per line of code, while still early in the course. Moreover, it seems that they instantly appreciate the fact that *they* need to take actions, in order to acquire space for the data of the processing specification (variable/constant declarations).

Students avoid using areas, which in the depiction are blank. Eventually, they implement an initialization step at an appropriate position in the program, in order to produce the desired PWS state.

Using the graphical representation, students differentiate between DASAs which are directly or indirectly referenced, and are able to produce the *reference-path*. Eventually, data structures are identified as rooted directed graphs, the nodes of which are DASAs.

Students question actions (and, therefore, instructions) that produce DASAs which a) in the depiction remain blank throughout the PWS states, and b) after initialization are only accessed once. Progressively, they attempt to maintain just the necessary DASAs in every PWS state.

The frequency of use of the PWS is high:

- Early in the course
- When control structures are introduced
- When data structures are involved
- When sort and search techniques are introduced
- When recursive calls are involved
- During debugging exercises

Students tend to use the PWS less frequently towards the end of the term, as well as, when the processing specification and the data constructs are rather simple.

It is necessary to point that weak students tend to use the PWS more often than other students. Furthermore, it has been noted that weak students, who regularly use the PWS, consistently outperform those who do not and gradually reach a satisfactory level of understanding of "programming practice".

[7] A common argument among educators is the inappropriateness of the use of a box to represent a variable.

The fact that students use the depiction in order to justify their answers underpins the significance of the graphical representation. The fact that the explanations that the students provide are in accordance with the addressed programming rules is interpreted as evidence of the suitability of the PWS as a representational system. Its success is primarily attributed to: a) the ability to refer to the dynamic changes of memory usage in terms of time states, and b) the explicit demonstration of the relations between the memory and the program's instructions.

5.1 Assessing the Mental Model

The use of the PWS aims to trigger the programmer's mentality. Is the emerged mental model viable? Is it valid? Is it *refinable*?

At the end of the term, the average student is able to theoretically explain the taught concepts, comprehend how concepts are used in given problems, apply them to exercises, and proceed satisfactorily with a short project which requires analysis of components, synthesis and generalization of ideas presented in class, and evaluation of the possible implementation alternatives. Reflecting the educational objectives of Bloom's taxonomy [8], it seems safe to claim that the emerged mental model is consistent with the intended mindset an introductory programming course aims to grow.

In their study, Fix et al. [17] identify the following characteristics in experts' mental representations of programs: a) they are hierarchically structured, b) they have explicit mappings between the layers, c) they use basic recurring patterns for program comprehension, d) they are well-connected, and e) they are well-grounded, in that they "include specific details of where structures and operations physically occur in the program". Based on the presented observations, it seems that a mental representation, generated by a user of the PWS, will be relatively consistent with the anticipated expert mentality, since a) the PWS is by nature hierarchical, b) its study requires explicit mappings between layers, c) it facilitates the detection of recurring patterns, d) it enforces the appreciation of components' interrelatedness, and e) every state, usually provides enough details to locate a certain operation in the program.

Finally, monitoring the progress of the students in the "Computer System Architecture" course, it proved they were able to efficiently reposition memory usage at the appropriate "hardware" locations, based on the attributes of implicitly vs. explicitly referenced storage areas.

6. CONCLUSION

Being abstract enough to fit the semantics of potentially any programming language, the PWS methodology may be integrated in the teaching/learning process of introductory programming, as a means to a) present several programming concepts, while revealing their interrelatedness, and b) address crucial pre-conceptions of novices.

A weak point of the system's current implementation media (traditional media, such as pen & paper) is that reproducing the PWS states may be quite "space-consuming" and, eventually, depending on the program's complexity, inefficient. Program animation tools, like Jeliot [34], may be productively used (to a certain extent) for support. Nevertheless, the development of tools, which will enable the users to *design* the PWS states and, thus, facilitate the major learning dimension of the methodology, is almost compulsory.

How constrained the representation should be, the exact implementation in the context of various programming languages, the level at which the emerged mental model suffices for the programming practice, and the ways it can be refined are some of the research questions that arise from this study.

The PWS is not, by any means, presented as a panacea for refining all possible preconceptions or dealing with all difficult concepts, like an introductory programming course does not aim to develop computing experts. The necessity of a viable computer model in the early stages of the study of any computing discipline is acknowledged in this thesis. However, due to facts and constrains presented in section 1, the suggestion is that its development takes place in the time/space frame of all introductory courses.

7. ACKNOWLEDGEMENTS

Many thanks to J. Kiourktsoglou, R. Cox, and R. Lutz for their valuable comments.

8. REFERENCES

[1] ACM/AIS/IEEE-Curriculum 2005 Task Force. *Computing Curricula 2005*. IEEE Computer Society Press and ACM Press, September 2005. (http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf)

[2] ACM/IEEE-Curriculum 2001 Task Force. *Computing Curricula 2001, Computer Science*. IEEE Computer Society Press and ACM Press, December 2001. (http://www.acm.org/education/cc2001/final/index.html)

[3] Ben-Ari, M. Bricolage Forever! In *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group*, University of Leeds, UK, 1999.

[4] Ben-Ari, M. Constructivism in Computer Science Education. In *Proceedings of the 29th SIGSCE Symposium*, Atlanta, USA, February 1998.

[5] Ben-Ari, M. *Understanding Programming Languages*. John Wiley & Sons, 1996. (http://stwww.weizmann.ac.il/G-CS/BENARI/books/)

[6] Blackwell, A.F. First Steps in Programming: A Rationale for Attention Investment Models. In *Proceedings of the IEEE Symposia of Human-Centric Computing Languages and Environments*, pp. 2-10, 2002.

[7] Blackwell, A.F., Green, T.R.G. Investment of Attention as an Analytic Approach to Cognitive Dimensions. In T. Green, R. Abdullah & P. Brna (Eds.), *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pp. 24-35, 1999.

[8] Bloom, B.S. *Taxonomy of Educational Objectives: The Classification of Educational Goals – Handbook 1: Cognitive Domain*. Longmans, 1965.

[9] Bonar, J. & Soloway, E. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1(2), pp. 133-161, 1985.

[10] Boudreau, G. & Pigeau, R. The Mental Representation and Processes of Spatial Deductive Reasoning with Diagrams and Sentences. *International Journal of Psychology*, 36(1), pp. 42-52, 2001.

[11] Braine, M.D.S. & O'Brien, D.P. A Theory of It: A Lexical Entry, Reasoning Program, and Pragmatic Principles. *Psychological Review*, 98, pp. 182-203, 1991.

[12] Braine, M.D.S. On the Relation between the Natural Logic of Reasoning and Standard Logic. *Psychological Review*, 85, pp.1-21, 1978.

[13] Cheng, P.C.-H. Unlocking Conceptual Learning in Mathematics and Science with Effective Representational Systems. *Computers in Education*, 33(2-3), pp. 109-130, 1999.

[14] DuBoulay, B. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), pp 57-73, 1986.

[15] Efopoulos, V., Dagdilelis, V., Evangelidis, G. Satratzemi, and M. WIPE: A Programming Environment for Novices. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Caparica, Portugal, 2005.

[16] Etmer, P.A. and Newby, T.J. Behaviorism, Cognitivism, Constructivism: Comparing Critical Features from an Instructional Perspective. *Performance Improvement Quarterly*, 6(4), pp. 50-70, 1993.

[17] Fix, V., Wiedenbeck, S, Scholtz, J. Mental Representations of Programs by Novices and Experts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, 1993.

[18] Gonzalez, G. Constructivism in an Introduction to Programming Computer Course. *Journal of Computing Science in Colleges*, 19(4), pp. 299-305, 2004.

[19] Green, T. Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities. In *Proceedings of the Working Conference of Advanced Visual Interfaces (AVI2000)*, Palermo, Italy, 2000.

[20] Haberman, B. & Kolikant, Y.B.D. Activating "Black Boxes" instead of Opening "Zippers" – a Method of Teaching Novices Basic CS Concepts. In *Proceedings of ITICSE 2001*, Canterbury, UK, pp. 41-44, 2001.

[21] Johnson-Laird, P.N. & Byrne, R.M.J. Precis of Deduction. *Behavioural and Brain Sciences*, 16, pp. 323-380, 1993.

[22] Johnson-Laird, P.N. *Mental Models: Towards a Cognitive Science of Language, Inference, and Conciousness*. Cambridge University Press, Cambridge, 1983.

[23] Jonassen, D.H. Objectivism versus Constructivism: Do We Need a New Philosophical Paradigm? *Educational Technology Research and Development*, 39(3), pp.5-14, 1991.

[24] Kuittinen, M. and Sajaniemi, J. Teaching Roles of Variables in Elementary Programming Courses. In *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, Leeds, UK, 2004.

[25] Lahtinen, E. & Ahoniemi, T. Visualizations to Support Programming on Different Levels of Cognitive Development. In *Proceedings of the 5th Koli Calling Conference on Computer Science Education*, Koli, Finland, November 17-20, 2005.

[26] Larkin, J.H., Simon, H.A. Why a Diagram Is (Sometimes) Worth a Thousand Words. *Cognitive Science*, 11, pp. 65-100, 1987.

[27] Lui, A.K., Kwan, R., Poon, M., Cheung, Y.H.Y. Saving Weak Programming Students: Applying Constructivism in a First Programming Course. *ACM SIGSCE Bulletin*, 36(2), pp. 72-76, 2004.

[28] Mavaddat, F. An Experiment in Teaching Programming Languages. *ACM SIGCSE Bulletin*, 8(2), pp. 45-59, 1976.

[29] Mayer, R.E. Different Problem-Solving Competencies established in Learning Computer Programming with and without Meaningful Models. *Journal of Educational Psychology*, 67, pp. 725-734, 1975.

[30] Mayers, B.A. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1, pp. 97-123, 1990.

[31] McCartney, R. and Sanders, K. What are the "Threshold Concepts" in Computer Science? In *Proceedings of the 5th Koli Calling Conference on Computer Science Education*, Koli, Finland, November 17-20, 2005.

[32] Meyer, J. & Land, R. Threshold Concepts and Troublesome Knowledge (2): Epistemological Considerations and a Conceptual Framework for Teaching and Learning. *Higher Education*, 49(3), pp. 725-734, 2005.

[33] Meyer, J. and Land, R. Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within Disciplines. *ETL Project Occasional Report 4*, Universities of Edinburgh, Coventry, and Durham, 2003.

[34] Moreno, A. & Myller, N. Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In *Proceedings of International Conference on Networked E-Learning for European Univarsities*, Granada, Spain, 2003.

[35] Pane, J.F. & Myers, B.A. Usability Issues in the Design of Novice Programming Systems. *Technical Report CMU-CS-96-132*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, USA, 1996.

[36] Pea, R.D. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, 2(11), pp. 25-36, 1986.

[37] Petre, M., Baecker, R, & Small, I. An Introduction to Software Visualization. In J. Stasko, J. Domingue, M.H. Brown, B.A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience*, MIT Press, pp. 3-26, 1998.

[38] Powers, K.D. Teaching Computer Architecture in Introductory Computing: Why? and How? In *Proceedings of the 6th Australasian Computing Education Conference (ACE2004)*, Dunedin, New Zealand, January 2004.

[39] Shu, N.C. Visual Programming: Perspectives and Approaches. *IBM Systems Journal*, 28(4), pp. 11-34, 1989 (reprinted 1999).

[40] Smith, J.P., diSessa, A.A., Roschelle, J. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The Journal of the Learning Sciences*, 3(2), pp. 115-163, 1993.

[41] Viner, M. Constructivism and Educational Implications for Teaching and Learning. *Journal of Educational Computing, Design & Telecommunications*, 3(1), 2003.

[42] VonGlasersfeld, E. A Constructivist Approach to Teaching. In Steffe and Gale (Eds.), *Constructivism in Education*, Hillsdale, NJ, Lawrence Elbaum Associates, pp. 3-15, 1995.

[43] White, G. Misconceptions in CIS Education. *Journal of Computing Sciences in Colleges*, 16(3), 2001.

[44] Wulf, T. Constructivist Approaches for Teaching Computer Programming. In *Proceedings of the 6th Conference on Information Technology Education (SIGITE '05)*, Newark, NJ, USA, 2005.

Moral Conflicts Perceived by Students of a Project Course

Tero Vartiainen
Turku School of Economics Pori
Unit
P.O. Box 170
FIN-28101 PORI, FINLAND
tero.vartiainen@tse.fi

ABSTRACT
Regardless of the popularity of project courses in computing curricula, little is researched on moral issues in these courses. The aim of this study was to increase understanding in this area by determining what students on a project course in information systems (IS) perceived as moral conflicts. Data was gathered from diaries, drawings, interviews and questionnaires, and the analysis was inspired by phenomenography. The results show that the hardest moral conflicts were confronted when a student acted in a project manager's role, and that many originated in problems related to the group process. A two-dimensional structure of moral conflicts was found. The results are considered in the light of the existing literature, and implications for research and practice are presented.

Keywords
project-based learning, moral conflicts, group process

1. INTRODUCTION
Project work is a commonly used work method in the IT field, and is considered an essential component in educating future computer professionals [12]. The benefits of project courses are evident in that students acquire communications skills [27], and team-building and interpersonal skills [29], for example. In cases in which student projects are implemented for real-life clients [32] rather than being purely hypothetical, students gain valuable experience for the start of their careers. Indeed, collaborative student projects are a common form of industry-academia collaboration in the IT field [36]. This kind of collaboration benefits industry by producing results and opening up contacts to students - who are possible future employees. Experiences from project courses show that there are ethical issues to deal with [8,30], but there are no in-depth studies about moral conflicts in IT projects in academia or in practice. This study aims to fill this gap by determining what students enrolled in a project course in information systems (IS) education perceive as moral conflicts.

In philosophical terms a moral dilemma is defined as a decision-making situation in which two incompatible actions are morally required [13, p.3]. A moral conflict is perceived as solvable and a moral dilemma as insolvable [15]. According to the broadest definition, a moral dilemma occurs in any situation in which morality is relevant [2]. In colloquial language the term moral problem is used, but for simplicity reasons, moral conflict is used in this empirical study. Examples of moral conflicts in computing, such as seeking balance between the quality of the information system and its cost, are found in Anderson et al [1].

This article is a partial report of a study investigating moral conflicts perceived by clients, students, and instructors on a project course [37]. Here, moral conflicts perceived by students are reported. Following this introduction, Section 2 reviews the literature related to the characteristics of small groups, group processes, and ethical issues in project courses. The research design is presented in Section 3, and the results in Section 4. Section 5 concludes the paper with some reflections on the results in the light of the literature.

2. LITERATURE REVIEW
A small group consists of three or more persons who are involved in social interaction aimed at achieving a common goal [4,16]. Indeed, the existence and all the actions of a group are geared towards the group goal [5]. Members share a set of values, they acquire or develop resources or skills, they conform to a set of norms, and they have a goal and leadership to coordinate their resources [14, p.4-5 and p.12-13]. The process of group decision-making includes task and social dimensions [9]. The task dimension refers to the relationship between the group members and the work they are to perform, and the social dimension to the relationships of the group members with one another. It seems that the task and social dimensions are highly interdependent. Group cohesiveness means the ability of the group members to get along with each other, thus determining their loyalty and commitment towards each other and the group, and could be seen as output from the group's social dimension. The output from the task dimension could be described as productivity. Cohesiveness and productivity are not easily determined, but they seem to have an interdependent relationship. The more cohesive the group is, the more productive it is. Cohesive groups are able to tolerate some differences in people, but too much variety in norms and values reduces the cohesiveness [9]. The above-mentioned task and social dimensions are visible in Tuckman's model of group development [33], which consists of forming, storming, norming, and performing stages. Later, Tuckman and Jensen [34] added the adjourning stage to the model (Table 1). At the forming stage, group members establish dependency relations with leaders and other group members, and pre-existing standards. They also test their boundaries of interpersonal and task behaviours. The storming stage involves conflict and polarization in relation to interpersonal issues as members resist the group influence and the task requirements. When resistance is overcome, at the norming stage, cohesiveness and in-group feelings develop, new standards evolve and new roles are adopted, and intimate and personal opinions are expressed. At the performing stage the interpersonal structure becomes the tool of task activities, and roles become flexible and functional: the group is targeted on the task. Finally, when the group's

work is completed at the adjourning stage, the members feel anxiety and sadness.

Table 1: Stages of group process [33, 34]

Stage	Characteristics
Forming (orientation, testing and dependence)	Testing and dependence; Orientation to the task
Storming (resistance to group influence and task requirements)	Intragroup conflict; Emotional response to task demands
Norming (openness to other group members)	In-group feeling and cohesiveness develop, new standards evolve and new roles are adopted; Open exchange of relevant interpretations, intimate, personal opinions are expressed
Performing (constructive action)	Roles become flexible and functional, structural issues have been resolved; Interpersonal structure becomes the tool of the task activities, group energy channelled to the task
Adjourning (disengagement)	Anxiety about separation, sadness; Self-evaluation

The group process theory together with common knowledge shows that group work may occasionally be turbulent and there are conflicts to deal with. This being the case it is surprising how rarely moral issues on project courses in computing are raised in computing literature. The following extract illustrates some problems in detail [30, p. 112]:

'… a) loners do not work well with others and want to "do their own thing", b) whistle blowing may not be done for various reasons, and c) handling the typical work ethic where "a few students do most of the work, some do just enough to get by and some do almost none."'

The above moral issues relate to individual students' acts and behaviour in the group. From the instructor's viewpoint, assigning meaningful grades is ethically difficult [30]. For example, whom should the instructor believe when a student or a team complains that another student or team is not doing its share of the work, or if a student complains that he or she would have been much more successful than the other students in accomplishing the project objectives?

Fielden [8] recalls experiences from over 10 years of conducting a student project course. Moral issues in student projects have emerged in the relationship between users and students when the users have had unrealistic expectations about what the students can accomplish. Problems also emerge when the student group does not come up with what was agreed when the contract with the client was signed. The same kinds of problems arise if one student from the group claims that it is able to accomplish something it cannot do (due to a lack of skills and expertise among the group members), or if there are different individual commitment levels. Dubious work practices in the client organization may also be cause for concern.

Ethics and morals relate to how individuals treat each other. As people in groups operate in close collaboration and as the stages of the group process theory show, there are conflicts to deal with. Therefore, students presumably face conflicts which have moral significance in their work. Next, the research design for increasing knowledge in this are is presented.

3. RESEARCH DESIGN

In order to identify moral conflicts in group-based project work and the same time to learn small group guidance work I started to work as the instructor. I applied participant observation [7] on an obligatory project course in IS curriculum in a Finnish university. Typically students participates the course in their third academic year and they represent ages between 20 and 25. Students on this course are put into groups of five to implement a project task defined by a client, typically an IT firm such as a software house, or the IT department of an organization, such as an industrial plant. Each student is expected to use 275 hours in implementing the project task, and 125 hours to demonstrate project-work skills, including project leading, group work and communication skills. In total, a five-student group uses 1,375 hours in planning and implementing the client project. Each student is expected to practice the role of project manager for about one month during the process, which lasts from five to six months. The tasks range from extreme coding projects to developmental projects and research. The tasks are typically ill-defined, and need to be redefined as the project progresses. A board consisting of two client representatives, two representatives from the student group (the project manager and the secretary) and the instructor make the redefinition and other decisions. During the collaboration, the role of the clients is to provide the students with guidance in terms of substance (e.g., technical guidance), and the role of the instructors is to guide the process (e.g., planning, reporting). Client organisations pay the university 8,500 euros for the co-operation. Once the supervising course instructor has accepted the clients, their representatives present the project tasks to the pre-formed student groups during the task-exhibition stage. The student groups then negotiate the tasks, and when clients and groups are allocated to each other the collaboration between a group and a client starts with an introductory meeting, the signing of the contract, and with the first board meeting.

During the project course, I arranged a voluntary ethics course for the students in order to develop their moral sensitivity and judgment [28]. I used student diaries, interviews, drawings and questionnaires, in which I encouraged the students to deliberate about moral conflicts they confront or could possibly confront. I encouraged them to express themselves openly in their own terms, an approach that allows complexities in moral actions to be taken into account [25, p.5]. Consequently, in line with the ideas of the interpretation theory [21], in order to reach understanding about moral conflicts the investigator has to interpret the subjects' statements to discover the meanings of their perceptions. The students were asked to use their diaries to reflect on the moral conflicts they confronted. During the first year (the academic period 1999-2000), the group members were required to write a common diary along with their personal diaries, but in later periods (2000-2001, 2001-2002) only personal diaries were used. In total, 13 individual students (coded S1…S13) and six student groups (coded G1…G6) wrote diaries (some of these groups were interviewed), 17 students (not involved in the ethics course) responded to a survey, and a total of 20 students produced drawings of moral conflicts during related exercises during the project course (some of these students later dropped out of the ethics course).

The analysis of the students' perceptions was inspired by phenomenography. The aim of the phenomenographical method is to identify and describe qualitative variation in individuals' experiences of their reality [23]. What is characteristic of the approach is that the aim is to capture conceptualisations that are faithful to the individuals' experiences of a selected phenomenon. These conceptions, which are typically gathered during interviews, are then categorised and relations between the categories are further explored [10]. A phenomenographic researcher seeks qualitatively different ways of experiencing the phenomena regardless of whether the differences are between or within individuals. He or she tries to achieve a so-called second-order perspective on the investigated aspect of the reality by describing the conceptions of a group of individuals - instead of taking the first-order approach and describing the reality directly, which is the convention in ethnographical studies (Figure 1) [18,35].

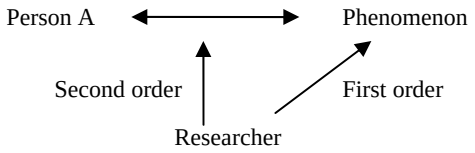

Figure 1: The first and second order perspectives [18,35]

In the analysis I coded the conflicts that emerged from the data (diaries, drawings, questionnaires) in order to acquire understanding of subjects' perceptions. Flap boards and the network views supplied in Atlas.ti software [24] helped me to visualize the categorization procedure. I grouped similar problems shared between the subjects, that is to say, I produced "pools" of moral conflicts. In this way I tried to achieve the second order perspective, abstractions of the issues towards what the subjects' deliberations were targeted. The separation to project task related issues and human issues became very clear: for example, deliberation about prioritizing between tasks and how a project manager should treat group members were logically allocated to different categories. While producing categorizations relating to the issues, I struggled with differences in the perceptions concerning the same issues. For example, regarding the human issues students deliberated on the one hand their own well-being and on the other hand well-being of the others. Moral psychology theories [20.26] led me to think about intentions behind the deliberations, and finally I concluded that the two dimensions describe the collective meaning structure of moral conflicts: the external and the internal dimensions. The dimensions and categorizations are presented in the following section.

4. RESULTS

The identified dimensions of moral conflict are presented first, and then the various categories are reported.

On the external dimension moral conflicts are divided into those involving outside parties, the project task, and human issues (the rows in Table 2). Outside parties relate to parties not involved in the particular project co-operation, but who are indirectly or directly influenced by it. Task-related moral problems refer to the attainment of objectives and the implementation of the tasks. The third group, human issues, relates to how individuals are treated in the project work. The identification of the internal dimension was based on objects of concern and care (the columns in Table 2). Here, the analysis was inspired by Piaget's [26] finding on children's development from egocentrism to perspective-taking, i.e., to the ability to take others into account. Taking care of others is perceived as more mature than only taking care of oneself. Thus, there are self-centred moral conflicts in which students are concerned about themselves. Although they may recognise other parties in their descriptions, they focus their concern and care on themselves and on their interests. Those experiencing other-directed moral conflicts extend their concern and care to others. Although the subjects themselves and others are present in the descriptions, and although self-centred interests are visible, the interests of others are dominant.

Category 1: Benefiting at the expense of outside parties. In this category, student deliberation is targeted on outside parties and is motivated by self-centred interests. While outside parties are recognised, duties and obligations towards them are not followed. There is a possibility of benefiting at the expense of outside parties by carrying out an illegal or harmful act.

Two themes emerged: the unauthorised copying of software and stabbing the other group in the back. First, producing unauthorised copies of software was considered a morally wrong act but it was nevertheless common: some students confessed that they had done unauthorized copying during the project. According to the student interviewees, the copying of installation CD-ROMs was usual in these groups. Secondly, one group may improve its own (and other groups') status by causing harm to another group. One student responded in the questionnaire that it was possible for students to stab another group in the back. The students were aware that every user in the university network was able to read other groups' documents – including the contract and the project results. It was revealed in the questionnaire responses that they could have reported the group to the instructors.

Category 2: Taking care of outside parties. In this category, student deliberation was targeted on outside parties and was motivated by concern for them. These parties include the whole of society, other groups, and people dependent on the client.

First, the students were concerned about societal problems and how they should relate to them. They perceived that their acts would have an influence on those issues. The problems in question included preserving natural resources, the questionable business line of a client, and intellectual property rights.

Group G2 considered the business line of their client questionable, for example. They felt that weaker people were in danger, and that employee burn-out was caused by assigning an unreasonable number of work tasks to them. The students were bewildered about the effects of decisions made by one individual: "The business line of the client of [name of the project group] is questionable. … one is able to destroy and seize firms that would be capable of surviving…. On the one hand, for us as a project group, do we want to work in favour of creating a society based on ownership and speculation?"

Secondly, there were references to maintaining relations with other groups and taking them into account. Other groups may be perceived differently and co-operation may be based on different assumptions. For example, student S13 analysed how she should relate to the other groups when she observed them stealing materials from the project space, and how she should react to her group members who spoke ill of other groups.

Table 2: Categories of moral conflicts perceived by students

The external dimension	The internal dimension	Self-centred Motivation and concern is based on the self.	Other-directed Motivation extends from self-centred deliberation to fulfilling one's duties and obligations and to concern for others.
Outside parties	Relations with parties outside the project group	Category 1: Benefiting at the expense of outside parties	Category 2: Taking care of outside parties
Project task	Attaining the objectives of the project and implementing the tasks.	Category 3: Self-centred deliberation related to the project task	Category 4: Fulfilling the project tasks
Human issues	Treatment of the individuals, who are participating in the project.	Category 5: Taking care of oneself and one's interests	Category 6: Taking care of the individuals in the project

Thirdly, concern was expressed about the division of the project tasks among the student groups. The clients present their project tasks to the students during the task-exhibition phase at the beginning of the course, and after that the students divide the tasks among themselves. The just allocation of the tasks was perceived as very difficult because many groups strove after the same ones, and it seemed that satisfactory resolution was impossible. Fourthly, students took into account parties that were dependent on representatives of clients. Those that emerged in the diaries were employees of the client organizations, and it was the possible harmful consequences of their acts to these parties that were of moral concern. Group G6 confronted a moral conflict when they analysed and wrote a report about their findings: protecting individuals' privacy would mean a loss of information from the final report. Getting useful results from the project was thus in conflict with protecting the employees of the organization. One student pondered on the effects of the results of their project on the employees of the client organisation in her drawing of a moral conflict: "Could we propose staff reduction as a result of our project work if the findings support this alternative?"

Category 3: Self-centred deliberation related to the project task. In this category student deliberation was targeted on the project task and was motivated by self-centred interests. Although other parties were recognised, obligations or duties towards them were not fulfilled. The category is based on students' observations concerning the inclination to avoid fulfilling one's duties, the use of university resources for one's own purposes, and carelessness in protecting confidential information.

First, the students felt that there was an inclination to avoid work tasks and to neglect the fulfilment of one's duties in the project work. Those in some groups observed other students avoiding tedious work tasks, guidance meetings and project managers' meetings. Student S8 stated that some of his fellow students tried to avoid work tasks: "I have noticed that within our group the level of commitment varies. They don't want to pick up the baton, and many of them seem keen to avoid work tasks." Secondly, moral conflicts were perceived in using university resources (e.g., copy machines) for purposes than other they were intended, namely for purposes based on self-interest. The students considered that the use of university resources such as copy machines, printers and telephones was included in the moral conflicts related to the inclination to use such resources for their own purposes. Thirdly, they confessed that they had conducted presumably or genuinely immoral acts in formal contexts: dishonesty in the booking of hours and plagiarism in project plans were mentioned. Student S12 stated that they were perhaps morally wrong when they copied the framework plans of groups from previous years. Group G1 confessed that they had falsified information in the booking of hours: "From time to time the project group added to the number of hours they booked in order to conform to the university time limits." Fourthly, students showed concern for the protection of confidential information, but they may have been careless in implementing their concern. It appeared from their statements that confidential information could leak to outside parties if emails fell into the wrong hands, or if they were talking about confidential issues in public places. According to student S9, adhering to a confidentiality agreement was hard in practice because discussions that started in the project room tended to continue in the university canteen, for example. Moreover, client issues were discussed more or less publicly at student parties: "Many times it has happened that the group has continued client-related discussions that started in the classroom in a canteen full of people. … The group members are obligated by the confidentiality contract, but following it seems to be hard in practice."

Category 4: Fulfilling the project tasks. Student deliberations in this category were targeted on the project task and motivated by concern for fulfilling the duties or obligations related to it. Although there were still self-centred concerns in the descriptions, there was also real concern about fulfilling duties and obligations for other parties. The following themes emerged: prioritising between project tasks and other issues, being careful about confidential information, complying with formalities in accordance with the rules, grading, and equal commitment to the project.

To start with prioritisation problems, students confronted moral conflicts related to the allocation of the time resources required for the work tasks and commitments such as other courses. Student S8 considered his position towards his fellow students' and his own time resources morally problematic. He tried to decide whether he should take care of the work tasks that others were not willing to do, and reflected upon this issue twice in his diary: "…I have plenty to do outside of the project, and I am not willing to sacrifice all my free time to it just because there is no agreement. Am I doing the wrong thing?" Student S6 observed that the aim to learn new technologies would not be most beneficial to the client, and using familiar equipment would be more efficient. The moral conflict in this case relates to the conflict between learning new skills and what is best for the client: "The group is obligated to produce a reasoned proposal about the implementation environment. Could the

group members' wishes affect the choice of environment – particularly if it would be undoubtedly useful for the client to use the environment about which the group has the best experience?"

The fact that there were unequal levels of commitment to the project was considered morally problematic in terms of student perceptions of how they reacted to such behaviour. Some of them stated in their questionnaire responses that there may be within group differences in commitment: some students are busier than others, and this affects the management of work tasks and timetables. As an example: "The Development Project course is very hard, and it is very tedious to complete other courses/to work simultaneously. It would be fair if all the group members allocated the same amount of their resources to the project - instead of taking on other work/courses to make their burden heavier and the scheduling more difficult" (questionnaire response). Figure 2 illustrates the same problem. The text in the drawing reads: "A group meeting. Where are the others?"

Figure 2. One student's drawing about a moral conflict.

Taking care of duties related to confidentiality was a concern for the students. The groups are encouraged to co-operate with each other by comparing their working methods and arranging brainstorming sessions, for example. Student S3 confronted a moral conflict when a student from their "synergy group" asked him what kind of education their client offered. The student did not know what he was allowed to talk about with other students, so he ignored the question. He faced a similar situation with his friends. The non-disclosure agreement forbade him from disclosing confidential information, but it was hard for him to understand what issues were included in the ban.

Students were bewildered about complying with project formalities in accordance with the rules. Should the compliance be strict, or was there some flexibility? The formalities in question were the booking of hours and the routines of board meetings. For example, student S8 noticed that the hours reserved for one phase had been exceeded, and he wondered whether he should reduce his own hours or tell the truth about having exceeded the allotted time. In any case, the booking of hours was problematic in terms of equality. The carrying out of work tasks may demand different amounts of time from students, i.e., there may be differences in efficiency, but still each one is required to use 375 hours for the project. Student S13 wrote in his diary about a disagreement with his fellow student over the booking of hours and the quality of his work: "… Of course, we others were not happy to do overtime, but someone had to do it to complete this project. Afterwards this feels unfair. [The name of the student] gave arguments for sticking to his guns: that he was more efficient than the others…."

The grading was considered morally problematic in itself because the assessment covered not only the group as a whole but also the individual members. The morality of giving different grades when every one's contribution was essential was raised - likewise the moral problem related to equal grading in cases in which all group members were not committed to co-operation. Student S2 deliberated about the problem of assigning different grades to group members: on the one hand each member was thought to play a significant role in the group, but on the other hand there were differences in commitment to the project, and such differences affected the group spirit. The following extract from student S2's diary reflects the problem: "… we started to produce the final assessment as group work. This raised a discussion about grades, because most of the group members thought that one of them did not deserve the same grade as the others. … Perhaps it is wrong to put group members in an unequal position after six months of work, particularly because everyone's contribution was unique … But it is self-evident that different working habits and schedules dampen the group spirit." (S2)

Category 5: Taking care of oneself and one's interests. Student deliberation in this category is targeted towards human issues and the motivation is self-seeking. Although the needs of other parties are recognised, the real concern is with oneself. Concern for one's wellbeing, serving one's own interests, and causing harm to another individual emerged as causes for concern.

The students in this category perceived a moral conflict in their own wellbeing in the project. Student S3 stated: "One must learn to say that one doesn't have time, and to be honest about one's abilities – otherwise one burns out." Conflict was also perceived in the pursuit of one's own interests and the disregard of those of the group. Students pursued their own interests by influencing the possibility of gaining employment from the client, for example. Group G1 stated in their joint diary that at the end of the project some of them were more concerned about their own interests, and disregarded the project. The possibility of securing employment from the client led them to advertise themselves to its representatives. One student pondered in her drawing of a moral conflict (Figure 3) about her behaviour towards a client representative in terms of gaining benefit from it. The text in the figure reads: "A moral conflict. Should I dance attendance on the disgusting client representative if I know that it will further my career? Double-dealing"

Bullying and harassment, which occasionally happened in the groups, could be interpreted as human-issues-related conflict: the student deliberates whether or not she will cause harm to another student. One student deliberated in her drawing on whether she could use her position as project manager to take revenge on her fellow student by assigning the most tedious work task to her: "Could I still use my position as a project manger to childishly take revenge [on my fellow student]?"

Figure 3: One student's drawing about moral conflict in project work

Category 6: Taking care of individuals. Student deliberation in this category is targeted on individuals and is motivated by

concern for other people's well-being or for fulfilling duties or obligations towards other individuals. Students referred to taking individuals into account in assigning work tasks, intervening in someone's actions, and honesty and ways of interacting with clients and instructors in their descriptions.

Students taking the role of project manager were concerned about the fellow-students to whom they assigned work tasks in terms of their ability to complete the tasks, their other activities that may be in conflict with the project tasks, and their efficiency. Student S2, in the project manager's role, confronted a moral conflict related to assigning a work task to a fellow-student whose ability to complete it was in doubt. On the one hand, he thought that, for the sake of honesty, he should probably tell the student of his concern, although the truth might hurt him. On the other hand, if he assigned the work task to him without taking any precautions, he might endanger the project. Another student produced two drawings (Figure 4). The first depicts a project manager ordering project workers about (with a whip in his hand), and a moral conflict between getting the job done and the group spirit emerges. The moral conflict is solved in the second drawing in that the project workers are having fun and at the same time they are able to produce results. In the left-hand picture the project manager is saying: "Make it snappy, you bastards! The deadline is drawing closer!!" His fellow students are complaining: "Ouch! We're tired but we have to work, Lord High and Mighty project manager." The text at the bottom of the figure reads as follows: "The final result vs. the group spirit (general wellbeing)" In the right-hand picture (representing the conflict as resolved) the project workers are saying: "Yeah! This is fun and the work is ready on time!" It could be concluded from the drawings that in attaining the final results a project manager may assign work tasks in either a repressive or a constructive way from the group-spirit perspective.

Some students confronted moral conflicts in which they had to think about whether they should intervene in another student's activities. The reasons for the possible interventions included the other student's irresponsible, ineffective, harmful, or evil behaviour. For example, in his role as a project manager, student S11 did not accept his fellow-student's not taking part in correcting the defects found in a document during inspection. He pondered on whether he should have intervened to change this student's behaviour.

Students perceived honesty-related moral conflicts in their co-operation with representatives of clients and university instructors. For example, student S2 speculated over whether refraining from telling a lie was the same as telling one. When his group presented a prototype of a future system they did not disclose all the problems they were struggling with.

Student S2 considered this a white lie that did not harm the project, and in the end they fixed the problems. Student S1 confessed that he was not fully honest in the final assessment because he did not reveal the real state of the relationship between him and another student (the other student had done something to offend student S1). S1 reasoned that, because he had a problem with only one student, and because problems with colleagues were not rare, he would not reveal his problem.

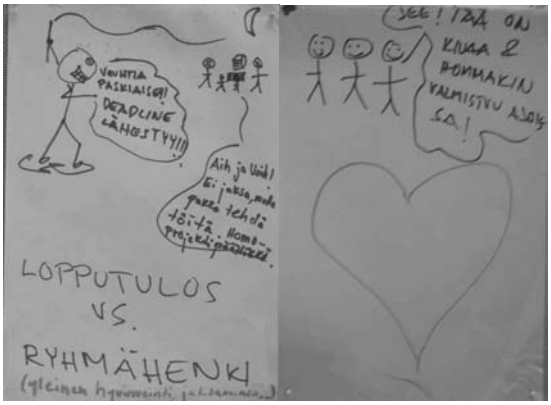

Figure 4. One student's drawing of a moral conflict in project work, and the conflict as solved

5. DISCUSSION

The results of this study increase understanding about moral conflicts in project work in a project course. The external division of moral conflicts into those related to outside parties, to project tasks, and to human issues well describes their themes. This division has its counterparts in the literature on group processes (task and social dimensions; [9]) and management. The classical managerial grid [3] consists of two aspects and their underlying concerns: management (concern for production) and leadership (concern for people). Indeed, students in the project manager's role had to tackle the hardest moral conflicts related to implementing the project task (concern for production), while at the same time upholding the group members' motivation (concern for people). Assigning work tasks and intervening in fellow-students' actions were, according to my interpretation, found by these novice managers to be the hardest moral conflicts. In addition to this, the students perceived that their projects had an indirect or direct influence on outside parties (e.g., employees of the client organisation). Indeed, the triplicity of external relations together with task and human issues has been recognised in the IS literature: project managers in information-systems development need skills in external relations together with task/project management and leadership skills [31], and project managers confront conflicts related to external stakeholders, managing the project (e.g., competition for scarce resources, differences related to goals) and interpersonal issues [17].

As far as these upper-level themes are concerned, the internal dimension uncovers the moral dimension as it denotes the intention behind the deliberation, which may be self-centred or other-directed. Students experiencing self-centred moral conflicts face temptations to break societal or group norms for egoistical reasons, such as getting software without paying for it and laziness in carrying out work duties. In these cases, when students are aware of that they break a norm or act against a moral value which they adhere to, their moral motivation, i.e. motivation to prioritize moral values above non-moral values [28], failed. However, not all self-centred moral conflicts relate to breaking a norm, and some involve concern for one's welfare (cf. upholding self in [11]). The interpretation adopted here, that egoism-based moral conflicts are forms of conflict perceived by the subjects, is supported by the results of studies on moral psychology, which recognise egoistical impulses as possible aspects (e.g., [25]). Students facing other-directed moral conflicts engage themselves in perspective taking, i.e., they are genuinely concerned about how the project work will affect outside parties, whether the duties and obligations relate

to the work tasks fulfilled, and how the group members are affected.

The results suggest that the developmental stage of group process in student groups may correlate with the severity and emergence of the moral conflicts confronted by their members. Many descriptions of such conflicts suggest similarities with the forming and storming stages [33] in the process of group development. Some of the self-centred conflicts encountered in this study indicate that not all of the group members were equally loyal or committed to the project task or to other members, given the noted avoidance of fulfilling one's duties and even harassment. As a consequence, other-directed moral conflicts arose in which project managers were forced to deliberate on how to intervene in the actions of group members showing this kind of behaviour. Building trust, a sense of togetherness and loyalty in these groups might have prevented these conflicts. Of course, individual student's sense of responsibility affect to his or her behaviour. It is suggested that relationship conflicts are more disruptive than task conflicts [6]. Presumably, groups experiencing human-issue problems are not as productive as groups with high cohesiveness. To sum up, these results suggest that the moral dimension (self-centred and other-directed concerns) is inherent in intra- and extra-group relations in student groups. Not all decision-making situations involve moral conflicts, but their emergence could be perceived as an inherent part of group work.

Implications for research and practice. Given the fact that the subjects of this study represent the Finnish population, similar research in other countries might reveal cultural differences. Although the project tasks were real, the research setting was an educational institute. There is thus a need for a similar study in a working-life setting. Other aspects of moral behaviour, and moral decision-making and the implementation of those decisions [20, 28], should also be investigated in the context of project work.

For educators, this study reveals moral conflicts students confront on project courses. As those assuming the project manager's role faced the hardest of these problems, it is suggested that students should be introduced to the leadership problems that beset those in managerial positions (e.g., [22]). In the case of the researched project course, students appeared to experience stress and anxiety in the collaborating with the outside client and other group members. Therefore, it is important to encourage students to take care of themselves and others. Additionally, ways of developing group cohesiveness should be introduced at the beginning of the course in order to foster the group process. This study offers examples on what happens in non-cohesive groups.

This study shows that practical project work is a fertile ground for ethics teaching. According to my practical experiences, it is possible to integrate ethics into project work. The external and internal dimensions of moral conflicts could be used as an instrument to develop students' moral sensitivity [28], and an introduction to ethics theory would assist them in the resolution process.

Evaluation of the study. The research is evaluated by principles put forward by [19] and the full description of the evaluation is to be found in [37]. Next, the principles with the most significant importance in relation to this study are considered.

First, the fundamental principle of hermeneutic circle is considered. The principle of hermeneutic circle is the basis for hermeneutics. This principle suggests that human understanding is achieved by iterating between the parts and the whole. In other words, we come to understand a complex whole from the meanings of its parts and their interrelationships and by iterating back and forth with interpretations until unresolved contradictions or gaps are filled. The principle of hermeneutic circle is actualised in this interpretive study by determining categories with external and internal dimensions, which constitute the second order perspective of students' perceptions.

Second, two principles, principle of interaction between the researchers and subjects, and the principle of suspicion are considered. The most influential source of bias in the data gathering was my presence and activities at various stages as a researcher, an instructor, and an ethics teacher. As ethics teacher I provided students with basic concepts relating to morals and ethics and directed them to deliberate about real moral conflicts they confront during the course. This may be considered both as strength and weakness of this study: the teaching intervention most probably steered students to deliberate issues, which they would not have otherwise deliberated. But from students' viewpoint fears of being shown up were significant, and therefore it is impossible to assess what they left untold, changed, or even invented in their expressions because of my triple role. In addition to this, as instructor I was to evaluate some students' performance. Before I started my instructor's job a student told me, "If you were the instructor, there might not be moral conflicts at all", suggesting that students would not be able to reveal such conflicts to their instructor. Although this statement is worthy of note, it turned out that the students described moral conflicts in detail in their diaries, and they sometimes expressed criticism of and frustration with the university, the instructors, the clients, and their fellow students alike.

Third, according to the principle of abstraction and generalization the researcher has to show how the abstractions and generalizations relate to the field study details. Although, in interpretive studies, very unique circumstances are investigated, these unique instances may be related to ideas and concepts, which apply to other situations. In the research design I reported how I collected and analysed data and in the results section the dimensions and categorizations are presented together with extracts from the data. Taken the issue of generalizing the results it is noteworthy that because this study is an in-depth case study by nature, the results are not directly generalizable to other project courses. However, the results point out some problem areas, which could be deliberated in other student projects courses – especially in the courses resembling the course I studied. The comparison with the relevant literature strengthens the view that the most significant features of moral conflicts in student projects are visible.

6. REFERENCES

[1] Anderson, R.E., Johnson, D.G., Gotterbarn, D., Perrolle, J. 1993. Using the New ACM Code of Ethics in Decision-Making. Communications of ACM 36 (2), 98-107.

[2] Audi, R. (Ed.) 1995. The Cambridge Dictionary of Philosophy. Cambridge: Cambridge University Press.

[3] Blake, R.R., Mouton, J.S. 1978. The New Managerial Grid. Houston: Gulf Publishing Company. Rererenced in F.E. Kast, J.E., Rosenzweig 1985. Organization & Management, A Systems and Contingency Approach. New York: McGraw-Hill.

[4] Boethius, S.B. 1983. Autonomy, Coping and Defense in Small Work Groups. Stockholm: Department of Psychology, University of Stockholm. Dissertation.

[5] Brown R. 2000. Group Processes. Oxford: Blackwell Publishers.

[6] De Dreu C.K.W. Weingart L.R. 2003 Task Versus Relationship Conflict, Team Performance, and Team Member Satisfaction: A Meta-Analysis. Journal of Applied Psychology 88 741-749.

[7] Fetterman, D.M. 1998. Ethnography: Step by Step. Thousand Oaks: Sage.

[8] Fielden, K. 1999. Starting Right: Ethical Education for Information Systems Developers. In C.R. Simpson (Ed.) AICEC99 Conference Proceedings, 14-16 July 1999. Melbourne. Brunswick East, Victoria: Australian Institute of Computer Ethics. 147-156.

[9] Fisher, B.A., Ellis, D. 1990. Small Group Decision Making, Communication and the Group Process. New York: McGraw Hill.

[10] Francis, H. 1993. Advancing Phenomenography. Nordisk Pedagogik 13 (2), 68-75.

[11] Gillian, W.R., Krebs, D.L. 2000. The construction of moral dilemmas in everyday life. Journal of Moral Education 29 (1), 5-22.

[12] Gorgone, J.T., Davis G.B., Valacich, J.S., Topi, H., Feinstein, D.L. Longenecker, H.E.Jr. 2002. IS 2002. Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems. Communications of AIS 11, Article 1.

[13] Gowans, C.W. 1987. The Debate on Moral Dilemmas In C.W. Gowans (Ed.) Moral Dilemmas. New York: Oxford University Press. 3-33.

[14] Hare, R.M. 1976. Handbook of small group research. New York: Free Press. Referenced in Hare, A.P., Blumberg, H.H., Davies, M.F., Kent, M.V. 1995. Small Group Research A Handbook. Norwood, New Jersey: Ablex Publishing Corporation.

[15] Hill, T.E. 1996. Moral Dilemmas, Gaps, and Residues: A Kantian Perspective. In H.E. Mason (Ed.) Moral Dilemmas and Moral Theory. New York: Oxford University Press. 167-198.

[16] Hollander, E.P. 1971. Principles and Methods of Social Psychology. New York: Oxford University Press. Referenced in Boethius, S.B. 1983. Autonomy, coping and defense in small work groups. Stockholm: Department of Psychology, University of Stockholm. Dissertation.

[17] Jurison, J. 1999. Software Project Management: the manager's view. Communications of Association for Information Systems. Vol 2, Article 17.

[18] Järvinen, P. 2001. On Research Methods. Tampere, Finland: Opinpaja Oy.

[19] Klein, H.K., Myers, M.D. 1999. A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. MIS Quarterly 23 (1), 67-94.

[20] Kohlberg, L. 1981. The Philosophy of Moral Development, Moral Stages and the Idea of Justice. San Francisco: Harper & Row.

[21] Little, D. 1991. Varieties of Social Explanation. An Introduction to the Philosophy of Social Science. Boulder: Westview Press.

[22] Manning, F.V. 1981. Managerial Dilemmas and Executive Growth. Reston, Virginia: Reston Publishing.

[23] Marton, F. 1986. Phenomenography – a research approach to investigating different understandings of reality. Journal of Thought. 21 (3), 28-49.

[24] Muhr, T. 1997. Atlas.ti. A software programme. Berlin: Science Software Development.

[25] Packer, M.J. 1985. The Structure of Moral Action: A Hermeneutic Study of Moral Conflict. Basel: Karger.

[26] Piaget, J. 1977. The Moral Judgement of the Child. Harmonsdsworth: Penguin.

[27] Pigford, D.V. 1992. The Documentation and Evaluation of Team-Oriented Database Projects. Proceedings of the twenty-third technical symposium on Computer science education. Kansas City, Missouri, United States. New York: ACM Press. 28-33.

[28] Rest, J. 1984. The Major Components of Morality. In W.M. Kurtines, J.L. Gewirtz (Eds.) Morality, Moral Behavior, and Moral Development. New York: A Wiley-Interscience Publication. 24-38.

[29] Roberts, E. 2000. Computing Education and the Information Technology Workforce. SIGCSE Bulletin 32 (2), 83-90.

[30] Scott, T.J., Tichenor, L.H., Bisland, R.B.Jr., Cross J.H. 2003. Team dynamics in student programming projects. SIGSCE 26 (1), 111-115.

[31] Semprevivo, P.C. 1980. Teams in Information Systems Development. New York: Yourdon Press.

[32] Tourunen, E. 1992. Educating reflective system designers by using the experiential learning mode. In B.Z. Barta, A. Goh, L. Lim (Eds.) Professional Development of Information Technology Professionals. Amsterdam: Elsevier Science Publishers. 113-120.

[33] Tuckman, B. W. (1965) "Developmental Sequence in Small Groups", Psychological Bulletin, Volume 63, Number 6, pp. 384 99, American Psychological Association.

[34] Tuckman B.W., Jensen M.A. (1977) Stages of small-group development revisited. Group Org. Studies 2: 419-427.

[35] Uljens, M. 1991. Phenomenography – A Qualitative Approach in Educational Research In L. Syrjälä, J. Merenheimo (Eds.) Kasvatustutkimuksen laadullisia lähestymistapoja. Kvalitatiivisten tutkimusmenetelmien seminaari Oulussa 11.-13.10.1990. Esitelmiä. Oulun yliopiston kasvatustieteiden tiedekunnan opetusmonisteita ja selosteita 39, 1991.

[36] Ziegler, W.L. 1983. Computer science education and industry: Preventing educational misalignment. In E.M. Awad (Ed.) The Proceedings of the Twentieth Annual Computer Personnel on Research Conference, Charlottesville, Virginia, USA. November, 10.

[37] Vartiainen T. Moral Conflicts in Project Course in Information Systems Education. Diss. Jyväskylä Studies in Computing 49. Jyväskylä: University of Jyväskylä.

System Papers

Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder

Petri Ihantola
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
petri@cs.hut.fi

ABSTRACT

Automatic assessment of programming exercises is typically based on testing approach. Most automatic assessment frameworks execute tests and evaluate test results automatically, but the test data generation is not automated. No matter that such test data generation techniques and tools are available.

We have researched how the Java PathFinder software model checker can be adopted to the specific needs of test data generation in automatic assessment. Practical problems considered are: How to derive test data directly from students' programs (*i.e.* without annotation) and how to visualize and how to abstract test data automatically for students? Interesting outcomes of our research are that with minor refinements *generalized symbolic execution with lazy initialization* (a test data generation algorithm implemented in PathFinder) can be used to construct test data directly from students' programs without annotation, and that intermediate results of the same algorithm can be used to provide novel visualizations of the test data.

Keywords

test data generation, symbolic execution, automatic assessment

1. INTRODUCTION

Besides software industry applications, typical examples where automated verification techniques are applied are numerous assessment systems widely used in computer science (CS) education (*e.g.* ACE [18] and TRAKLA2 [14]) – especially in systems used for automatic assessment of programming exercises (*e.g.* ASSYST [10], Ceilidh [4], and SchemeRobo [17]). Automatic assessment of programming exercises is typically based on testing approach and seldom on deducting the functional behavior directly from the source code (such as static analysis in [19]). In addition, it is possible to verify features that are not directly related to the functionality. For example, a system called Style++ [1] evaluates the programming style of students' C++ programs. The focus of this work is on testing, not on formal assessment. Therefore, the term *automatic assessment* is later on used for test driven assessment of programming exercises.

Testing is used to increase trust about the correctness of a program by executing it with different inputs. Thus, the first thing to do is to select a representative set of inputs. The input for a single test is called *test data*. After the test data has been selected, the correctness of the behavior of the program is evaluated. The functionality that decides whether the behavior is correct or not is called a *test oracle*. The test data and the corresponding oracle together are called a *test case*. Finally, the test data of a logical group of tests together are called a *test set*.

Test data generation can be extremely labor intensive. Therefore, automated methods for the process have been studied for decades (*e.g.* [6]). However, for some reason such systems are seldom used in automatic assessment.

In this work we will apply an automatic test data generation tool, namely Java PathFinder (JPF) [21], in test data generation for automatic assessment. In addition to previously reported techniques of using JPF in test data generation, we will improve these techniques further on. We will also explain how to automatically provide abstract visualizations from automatically generated tests.

Although we discuss automatic assessment and feedback, the core of this research is on automatic test data generation and visualization. We will introduce a new technology we hope to be useful in programming education. However, we are not yet interested in evaluating the educational impact of this work. Manual test data generation is already the dominant assessment approach in programming education. This work makes test data generation easier for a teacher and provides visualizations and better test adequacy for students. Thus, we believe that results of this work are valuable as is.

The rest of this article is organized as follows: Section 2 is about the previous research of others and heavily based on previous work of Willem Visser, Corina Păsăreanu, Sarfraz Khurshid, and others [2, 5, 12, 16, 20, 21]. Section 3 describes our contribution and changes to the previous techniques. Section 4 discusses about some quality aspects in different test data generation techniques and Section 5, finally, concludes the work.

2. AUTOMATIC TEST DATA GENERATION

2.1 Different Approaches

There are several different techniques for automated test input generation. There are also many test generation tools for programs manipulating references introduced in the literature (*e.g.* [3, 11, 22]). Unfortunately most test input generation tools are either commercial or unavailable for some other reason. In addition, many open source testing tools[1] concentrate on other aspects of testing than test input generation. Here we will not describe tools, but some techniques for test data generation. Later in Section 2.2, we will explain how the techniques can be implemented in JPF.

[1] `http://opensourcetesting.org/` [April 10, 2006]

2.1.1 Method Sequences vs. State Exploration

In unit testing of Java programs, test input consists of two parts: 1) explicit arguments for the method and 2) current state of the object (*i.e.* implicit `this` pointer given as an argument). The first decision in test input generation is to decide how object states are constructed and presented. There are at least two approaches to the task:

Method sequence exploration is based on the fact that all legal inputs are results from a sequence of method calls. A test input is represented as a method sequence (beginning from a constructor call) leading to the state representing test data.

Direct state exploration tries to enumerate different (legal) input structures directly (*i.e.* without using the methods of the class in the state construction). Heuristics can also be applied or the state enumeration can be derived from the control flow of the method to be tested (as in Section 2.2.3).

The common justification for using method sequence exploration is that in assessment frameworks, object states can only be constructed through sequential method calls. On the other hand, in the method sequence exploration, tests are no longer testing only a single method. If the methods needed in the state construction are buggy, it is difficult to test other methods. However, in automatic assessment, one might want to give feedback from all the methods of the class at the same time – not to say that feedback from method X cannot be given before problems in method Y are solved.

2.1.2 Symbolic Execution

The main idea behind *symbolic execution* [13] is to use symbolic values and variable substitution instead of *real execution* and real values (*e.g.* integers). In symbolic execution, return values and values of variables of programs are symbolic expressions consisting of symbolic input. For example, the output for a program like "`int sum(int x, int y) { return x+y; }`" with symbolic input `a` and `b` would be `a + b`.

A state in symbolic execution consists of (symbolic) values of program variables, a path condition and the program counter (*i.e.* information where the execution is in the program). Path condition is a boolean formula (or the corresponding constraint satisfaction problem (CSP)) over input variables and describes which conditions must be true in the state. A *symbolic execution tree* can be used to characterize all execution paths (*i.e.* state chains). Moreover, a finite symbolic execution tree can represent an infinite number of real executions. Formally, a symbolic execution tree SYM($\mathcal{P}$) of a program $\mathcal{P}$, is a (possibly infinite) tree where nodes are symbolic states of the program and arcs are possible state transitions.

For example, the symbolic execution tree of Program 1, `min( X, Y )`, is illustrated in Figure 1. In the initial state, input variables have the values specified by the (symbolic) method call and the path condition is *true*. Nodes with an unsatisfiable path condition are pruned from the tree (labeled "backtrack" in the figure).

All the leaf nodes of a symbolic execution tree where the path condition is satisfiable represent different execution paths. Moreover, all feasible execution paths of $\mathcal{P}$ are represented in SYM($\mathcal{P}$). In the example of Figure 1, there are two satisfiable leafs and therefore exactly two different execution paths in Program 1. All satisfiable valuations for a path condition of a single leaf node in SYM($\mathcal{P}$) will give us inputs with identical execution paths in the program ($\mathcal{P}$). Furthermore, all leaf nodes represent different execution paths. Thus, if SYM($\mathcal{P}$) is finite we can easily generate inputs for all possible execution paths in ($\mathcal{P}$) and if SYM($\mathcal{P}$) is infinite the maximal path coverage is unreachable.

The golden age of symbolic execution goes back to 70's. The original idea was not developed for the test data generation, but formal verification and enhancement of program understanding through symbolic debugging. However, the approach had many problems [7] including: 1) Symbolic expressions quickly turn complex; 2) Handling complex data structures is difficult; 3) Loops dependent on input variables are difficult to handle.

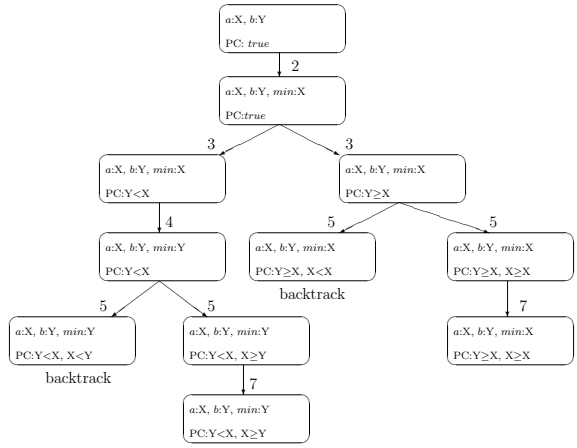

Figure 1: Symbolic execution tree of the Program 1. Numbers in the figure are line numbers.

```
1  int min( int a, int b ) {
2    int min = a;
3    if ( b < min )
4      min = b;
5    if ( a < min )
6      min = a;
7    return min;
8  }
```

Program 1: A program calculating minimum of two arguments. Line 6 is dead code (*i.e.* never executed) as one can see from Figure 1.

2.2 Test Data Generation with JPF

JPF is an open source explicit-state model checker of Java programs. Under the hood it is a tailored virtual machine, and therefore any compiled Java program (*i.e.* byte-code) can be directly used as an input for it. No source-to-source translation is needed as with many other model checkers.

In addition to standard Java libraries, JPF provides some library classes to control the model checking directly from the verified program. The following methods of the Verify class will be applied later in different test data generation strategies:

random(int n) will nondeterministically return an integer from $\{0, 1, \ldots n\}$.

randomBoolean() will nondeterministically return `true` or `false` nopagebreak

ignoreIf(boolean b) will cause the model checker to backtrack if b evaluates to true. The method is typically used to prune some execution branches away.

The fundamental idea behind nondeterministic functions is that whenever they are model checked, all the possible values are tried one by one.

JPF provides also a symbolic execution library. The library is not yet publicly available but an evaluation version was obtained for our study. The library provides types like SymbolicInteger, SymbolicBoolean and SymbolicArray. The main idea with the library is to provide model level abstractions for programmers. For example, integer variables are replaced with SymbolicIntegers and operators between integers with methods of the SymbolicInteger class

The symbolic library of JPF keeps track of the path condition. Whenever branching depending on a symbolic variable occurs (*i.e.* some of the comparison methods are called), the execution nondeterministically splits into two, and the condition (or its negation on the else branch) is added to the path condition. The framework uses a standard CSP solver for two tasks:

- Whenever a new constraint is added to the path condition, satisfiability is checked. If the path condition is unsatisfiable, `Verify.ignoreIf(true)` is called and the corresponding execution branch is pruned as the JPF backtracks.
- To provide concrete valuations for (symbolic) input states (*i.e.* to get concrete test data from a symbolic state)

2.2.1 Explicit Method Sequence Exploration

Explicit method sequence exploration is based on generating method sequences of different length by using the nondeterministic functions of JPF as in Program 2. The example is a container where states are constructed with insert and delete methods. The example generates all the method sequences up to 10 calls with arguments varying between 0 and 5. Actually, all the possible states of a traditional binary search tree can be constructed by repeating the insert method only, but all the states of the class are not necessarily reached with the same approach. For example, if a binary search tree uses lazy delete, all the states cannot be reached through inserts only.

2.2.2 Symbolic Method Sequence Exploration

Symbolic method sequence exploration is similar to explicit method sequence exploration. The only difference is that symbolic variables are used instead of concrete ones. Program 3 does the same as Program 2, but with symbolic values. Because arguments given for the BinarySearchTree are no longer integers but symbolic integers, the original container class needs to be annotated before the symbolic approach can be used. The annotation means that integers are replaced with SymbolicIntegers and operators with the corresponding method calls.

2.2.3 Generalized Symbolic Execution with Lazy Initialization

Generalized symbolic execution with lazy initialization, described by Visser *et al.* [12, 21] is a symbolic state exploration technique. In contrast to method sequence exploration, the approach does not require a priori bounds of the input structures (*e.g.* `END_CRITERIA` in Programs 2 and 3). Ideally the approach uses only the method to be tested in the test data generation. Thus, test data can also be generated to methods in a partially implemented class with some relevant methods missing.

```
1  public static final int END_CRITERIA = 10;
2  public static final int MAX_ARGUMENT = 5;
3  public static void main(String[] args) {
4    Container c = new BinarySearchTree();
5    for ( int i = 0; i <= END_CRITERIA; i++ ) {
6      if ( Verify.randomBoolean() ) break;
7      if ( Verify.randomBoolean() )
8        c.delete( Verify.random(MAX_ARGUMENT) );
9      else
10       c.insert( Verify.random(MAX_ARGUMENT) );
11   }
12 }
```

Program 2: Test data creation with explicit method sequence exploration for a `BinarySearchTree` class.

```
1  public static final int END_CRITERIA = 10;
2  private static void main(String[] args) {
3    Container c = new BinarySearchTree();
4    for ( int i = 0; i <= END_CRITERIA; i++ ) {
5      if ( Verify.randomBoolean() ) break;
6      if ( Verify.randomBoolean() )
7        c.delete( new SymbolicInteger() );
8      else
9        c.insert( new SymbolicInteger() );
10   }
11 }
```

Program 3: Test data creation with symbolic method sequence exploration for the annotated `BinarySearchTree` class.

The program to be tested is annotated so that fields are lazily initialized when they are first used. Special getters and setters have to be written for each field of the class. After that, fields are used through these methods only. When an unused (no previous reads or writes) field of a reference type is accessed through a getter, the field is nondeterministically initialized to any of the following:

- `null`
- a new object with uninitialized fields
- a reference pointing to any of the previously created objects of the same type (or subtype)

Primitive fields are always initialized to a new symbolic variable.

Method `_new_Node` in Program 4 (starting from line 8) is an example from such nondeterministic initialization. The method is called from the corresponding getter (*i.e.* `_get_next` starting from line 15). In `_new_Node`, the vector `v` contains the null object and all the objects created so far. The nondeterministic branching to select any item from `v`, or a completely new object, is on line 9.

Test data generation is launched by calling the method to be tested with an empty `this` object as argument. The empty object means an object with uninitialized fields. In the following, we will assume that `this` is the only reference argument, but other reference arguments would be handled similarly.

```
1  public class Node {
2    Expression elem;
3    Node next;
4    boolean _next_is_initialized = false;
5    boolean _elem_is_initialized = false;
6    static Vector v = new Vector();
7    static {v.add(null);}
8    Node _new_Node() {
9      int i = Verify.random(v.size());
10     if(i<v.size()) return (Node)v.elementAt(i);
11     Node n = new Node();
12     v.add(n);
13     return n;
14   }
15   Node _get_next() {
16     if(!_next_is_initialized) {
17       _next_is_initialized=true;
18       next = Node._new_Node();
19       Verify.ignoreIf(!precondition());//e.g. acyclic
20     }
21     return next;
22   }
23   Expression _get_elem() {
24     if(!_elem_is_initialized) {
25       _elem_is_initialized=true;
26       elem = new SymbolicInteger();
27       Verify.ignoreIf(!precondition());//e.g. acyclic
28     }
29     return next;
30   }
31   Node swap() {
32     if (_get_next() != null &&
33        _get_elem()._gt(_get_next().get_elem())) {
34       Node temp = _get_next();
35       _set_next(temp._get_next());
36       temp._set_next(this);
37       return temp;
38     } return this;
39   }
40 }
```

Program 4: Excerpts from an annotated program

```
1  public class Node {
2    int elem;
3    Node next;
4
5    Node swap() {
6      if ( next != null && elem > next.elem ) {
7        Node temp = next;
8        next = temp.next;
9        temp.next = this;
10       return temp;
11     }
12     return this;
13   }
14 }
```

Program 5: Example program

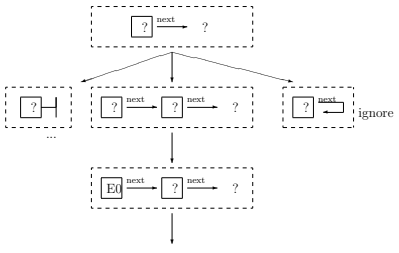

Figure 2: Excerpts from the symbolic execution tree of the Program 4, quoted from [12].

A problem is that lazy initialization can lead into illegal input structures. Thus, therefore a conservative class invariant is required. The invariant is implemented as a method that can determine if a (partially) complete object graph can be completed into a legal one. Actually such a precondition for each method separately would be sufficient. However, if an invariant can be defined, it can be used with all the methods of the class. Execution will backtrack if the invariant does not hold after the lazy initialization (see line 19 in Program 4).

Program 4 is the annotated version of Program 5. The example is quoted from Khursid *et al.* [12] with the annotation format quoted from Visser *et al.* [21]. The `precondition` method, which is not shown, is the class invariant that would return false if there is a loop in the list.

A partial symbolic execution tree of the program is provided in Figure 2. Only some first branches from the tree are taken into the figure. A question mark "?" inside a box is for an uninitialized value (*i.e.* elem field), but otherwise stands for an uninitialized reference (*i.e.* next field). In the initial state, the object for which the swap is called is created, but the fields (*i.e.* elem and next) are uninitialized. The figure demonstrates how new objects (*i.e.* list nodes and data objects in nodes) are created by the lazy initialization as the execution goes on. The first lazy initialization results from line 32 (Program 4). Evaluating "`_get_next() != null`" will result in the lazy initialization of the next field. The next will be initialized to any of the three possible cases, as illustrated by the first two rows of the figure.

What lazy initialization with symbolic values actually does, is generating the symbolic execution tree of the program. If the tree is finite, the approach will find all the leaf nodes of the tree, and therefore generate a test set with maximal path coverage [8]. However, if SYM($\mathcal{P}$) is infinite, the test data generation process does not terminate. One possible solution is to modify the JPF virtual machine so that only paths up to given length are checked. Another possibility is to set an upper limit for structure sizes in the class invariant. However, deriving actual test data from partially initialized object graphs is still an open problem. The constraint solver behind JPF will instantiate all the symbolic variables, but the unknown references are the problem. A simple solution is to make unknown references pointing to a special node called "unknown". Thus, graphs are not actually completed, but this should not be a problem because references pointing to "unknown" are not to be used as long as the program to be tested and the program to be used in the test generation are the same.

3. OUR APPROACH

Whereas the previous section was about related research of others, this section is about our own contributions to visualize test data and refine the symbolic execution based test data generation approaches of Section 2.

3.1 Visualizations for Abstract Feedback

Conceptually, in the approach we are now proposing, the outcome of automatic test data generation is not only a test set, but a set of *test patterns*. Each test pattern defines test data that are somehow similar. Test pattern is a kind of opposite to test set because the latter contains different test data in order to provide good test coverage. A possible grouping criteria for test patterns is that the

execution paths in the program are identical. In detail, a test pattern consists of a single *test schema* and possibly several test data derived from the schema. All the test data in the same test pattern are derived from the schema of the pattern. Finally, the test set is obtained by selecting arbitrary test data from each test pattern.

The schema is an object graph with two special features: 1) object references can be unknown and 2) symbolic expressions are used for primitive fields. In addition, the schema has constraints related to the symbolic expressions.

The schemas will be used to demonstrate tests on a higher abstraction level when compared to the actual test data. To understand the use of schema in the feedback, let us consider test schema $\mathbf{s}$ and test data $\mathbf{t}$ derived from $\mathbf{s}$. Instead of exact feedback saying $\mathcal{P}(\mathbf{t})$ fails (or works correctly), we will provide abstract feedback like "$\mathcal{P}(\mathbf{s})$ fails (or works correctly)". However, the oracle of the automatic assessment is based on investigating $\mathcal{P}_{\text{specification}}(\mathbf{t}) = \mathcal{P}_{\text{candidate}}(\mathbf{t})$, as in the traditional approach. Figure 3 illustrates this process and the related terminology.

Because test schema is an object graph with some constraints, it can be easily visualized. Figure 4, for example, provides visualizations from partially initialized object graphs of a delete method in a binary search tree. These example schemas were obtained after generalized symbolic execution with lazy initialization. Figure 5, on the other hand, gives examples from the possible test data that can be derived from the schema of Figure 4.

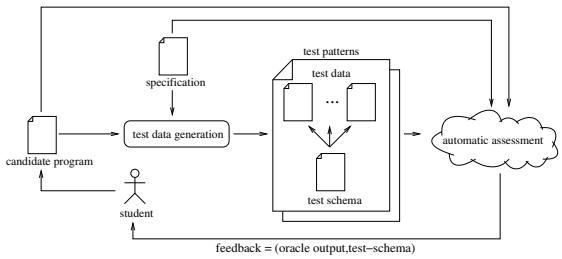

Figure 3: The process of creating feedback for students and some related terminology.

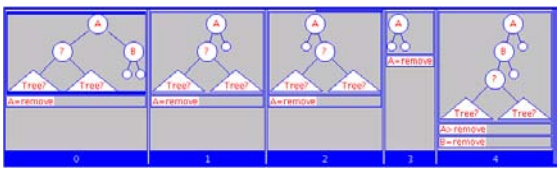

Figure 4: Excerpts of different input structures for the delete method of binary search trees

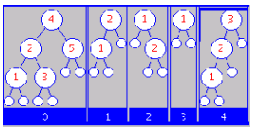

Figure 5: Examples of instantiated input structures from schemas in Figure 4

There are four types of nodes in the schema visualization: null nodes (small empty circles), nodes that are known to exist, but the data of the node is newer used (circles with ?), nodes with a data element that is used by the algorithm (circle with a letter), and nodes that represent a reference that is not used by the method (triangular nodes). In nodes where the data is used, keys inside nodes are symbolic variables and constraints over those variables are also provided.

3.2 Without Annotation

When deriving tests from students' programs, the manual annotation is not acceptable. This is because in automatic assessment the test data is generated on-the-fly, whenever a student submits a solution. Two techniques to remove the need of annotation in different use cases will be introduced: 1) use of the Comparable interface 2) A common upper class to a candidate program and the specification, called a probe.

3.2.1 Comparable Interface

Use of the Comparable interface can remove the need of replacing int type with SymbolicInteger. For example, let us assume a container implementation without primitive fields and where the data stored implements the Comparable interface. The interface is a standard Java interface used with objects having a total order. If the argument type in insert and remove methods of the container is Comparable, we can introduce the symbolic execution by using a special object that is comparable and hides the symbolic execution (Program 6). Moreover, students do not need this special class because they can test their container implementations, for example, with Integer wrappers.

```
1  public class ComparableSymbolicInt extends
       SymbolicInteger implements Comparable {
2    public int compareTo(Object other) {
3      ComparableSymbolicInt o = (ComparableSymbolicInt)
         other;
4      if (this._LT(o)) return -1;
5      else if (this._GT(o)) return 1;
6      else return 0;
7    }
8  }
```

Program 6: Definition of a comparable type that can hide symbolic execution so that programmers should not need to care about that.

The drawback of the Comparable approach is that in the comparison the execution will split into three when comparing SymbolicIntegers would split the execution into two. We will come back to this in Section 4.3.

3.2.2 Probes to Hide Invariants

Probes contain the specialized getters and setters of the lazy initialization as well as the class invariant. This makes it possible that those are not needed in classes derived from the probe. Thus, on-the-fly test generation from students programs is basically possible. The behavior of the getters and setters can be controlled so that lazy initialization can turned on and off.

The obvious limitation of probes is that new fields cannot be declared in a class derived from the probe. If new fields would be declared, lazy initialization of those would not be possible. This is mainly because the invariant method (declared in the probe) cannot say anything about the new

fields. Probes can be easily applied to exercises dealing with exactly defined data structures (*e.g.* implement the red-black-tree or implement the AVL-tree). The problem is how to handle more open assignments where the structure of the class can vary from solution to solution. Most likely, the probe approach cannot be used in such cases.

4. DISCUSSION

Three fundamentally different approaches of using JPF were described in Section 2.2: 1) explicit method sequence exploration 2) symbolic method sequence exploration, and 3) generalized symbolic execution with lazy initialization. In Section 3.2, we introduced two new approaches: the Comparable interface and probes. The new techniques are designed to help test data generation directly from students' candidate programs. New techniques can be combined with previous techniques and Figure 6 summarizes the resulting six[2] different test data generation approaches.

On the upper level, we have separated techniques between *method sequence exploration* and *lazy initialization*. On the second level, symbolic execution is used with lazy initialization but it can also be used in (symbolic) method sequence exploration. Symbolic execution and lazy initialization both need different types of annotations. New techniques we have developed are hiding these annotations from users. The Comparable interface answers to the challenge of symbolic execution and probes to the lazy initialization specific problems.

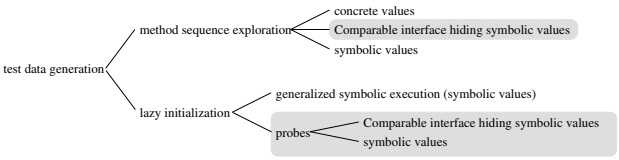

Figure 6: Different test data generation approaches under discussion: new techniques introduced in this work are on gray background, whereas techniques on white background are from the related (previous) research.

4.1 Preparative Work

Table 1: Evaluating the annotations needed

technique		annotation
Method sequences	with concrete	none
	with comparables	none
	with symbolic	automatic
Lazy initialization	generalized symbolic	semiautomatic
	probes with comparables	none
	probes with symbolic	automatic

The preparative work is evaluated based on the amount of annotations required. The scale includes values *none*, *automatic*, and *semi-automatic*. None means that absolutely no annotation is needed, automatic means that the annotation process can be automated, and semiautomatic means that the annotation process can be partially automated, but substantial amount of manual work is still needed. Table 1 summarizes our observations in this category.

[2]Not all combinations are reasonable.

Method sequence exploration requires some annotation if symbolic arguments are not hidden behind the Comparable interface. However, the annotation process can easily be automated as variables of int type are only replaced with SymbolicInteger variables and operations between integers are replaces with method calls. Visser *et al.* [21] have already described a semiautomatic tool for the task. Actually the tool can also construct additional fields needed in generalized symbolic execution with lazy initialization as well as getters and setters for fields. Use of fields are also replaced with getter calls and definitions (*i.e.* assignments) with calls to corresponding setters. The only task in the tool that is not automated is the type analysis.

The annotation that cannot be automated in generalized symbolic execution with lazy initialization is the construction of invariants or preconditions. However, probes can be used to hide invariants and other needs of annotation – just like Comparable hides simple use symbolic integers. The framework can provide support for common data structures and algorithms. For more exotic classes, a teacher can implement the probe for students. In both cases, if a probe is available, handmade annotations are not needed.

4.2 Generality

Table 2: Evaluating the generality

technique		generality
Method sequences	with concrete	*****
	with comparables	**
	with symbolic	****
Lazy initialization	generalized symbolic	****
	probes with comparables	*
	probes with symbolic	***

Generality is about what kind of programs can be used as a basis in test data generation. Table 2 gives relative ranking between techniques – more stars in the figure indicate that there are more situations in which the technique can be applied.

Method sequence exploration has practically no limitations and is therefore ranked at the highest place. The other techniques are first ranked according to the comparable *vs.* symbolic classification. Use of symbolic objects is considered a more general approach when compared to Comparables. The secondary classification criteria is the use of probes. If probes are not needed, it is considered more general when compared to cases where the program is built on probes.

In the concrete method sequence exploration, all the possible operations with arguments (*i.e.* integers) are directly supported. Bit level operations are also supported and data flow can go from input variables to other methods (*e.g.* to libraty methods that are difficult to annotate). The symbolic execution framework of JPF does not support bit level operations. In addition, data flow from test data to other methods is problematic in symbolic execution. Such an attempt would require the same preparative work for other methods, as well. For library methods, this might be extremely tricky. However, limiting the program to Comparables is considered a more significant drawback when compared to the limitations of symbolic integers. There are many practical examples when a simple program needs integer arguments, and the computation cannot be performed with comparable arguments only.

Probes can also limit the generality. A new probe is needed for every possible data structure, which limits the number of supported programs.

4.3 Abstract Feedback

Table 3: Evaluating the abstractness of schemas

technique		abstractness
Method sequences	with concrete	*
	with comparables	**
	with symbolic	**
Lazy initialization	generalized symbolic	***
	probes with comparables	***
	probes with symbolic	***

According to Mitrovic and Ohlsson [15], too exact feedback can make learners passive and therefore abstract feedback should be preferred. Correspondingly, on introductory programming courses at the Helsinki University of Technology, we have observed that exact feedback (*i.e.* "program fails where $a = 2, b = 4$") guides some students to fix the counter example only. After "fixing the problem", the candidate program might work with $a = 2$ and $b = 4$, but not with other values $a < b$.

In this category, the evaluation is based on how much test data can be derived from the same test schema. In other words, how general is the schema. All the described approaches have a property that executions leading into two different execution paths cannot be derived from the same schema. Table 3 gives the relative ranking between techniques – more stars in the figure indicate that the schemas are more general.

Concrete method sequence exploration is the least abstract method because the schema and test data are the same. Lazy initialization is the most abstract approach as test schemas with it are only partially initialized object graphs. For each partially initialized symbolic graph, there are (several) symbolic graphs that can be obtained through method sequence exploration.

Another aspect related to the abstractness of schemas is redundancy. We have defined schemas so that all the test data derived from a single schema will lead into identical execution paths. However, it is possible that there are several schemas stressing one path only. This is what we call redundancy. Therefore, the more abstract the schema is, the less redundant it is.

A reason why the concept of redundancy is interesting is that even with the most abstract approaches some redundancy exists. The extra branching, and therefore redundancy, that the Comparable interface brings was described in the previous chapter. When comparison of symbolic integers has two possible values (a < b is either true or not) the comparison of Comparable objects had three possible outcomes (less than, equal, and greater than).

Nondeterministic branching in lazy initialization will also add extra branching to the program. Let us think about binary search tree delete operation. If the node to be deleted has two children, the minimum from the right subtree will be spliced out as in Program 7 which is an excerpt from the delete routine. Both input structures in Figure 7 are obtained through the lazy initialization with probes. The node to be deleted is A in the both cases. In both cases, B is the smallest value in the right subtree of A. Thus, B is spliced out, by setting the link (originally pointing to B) to the right child of B. The right child of B is accessed. As a consequence, it is initialized to `null` or a new object. Because the `right` pointer in A is simply set to the right child of B, the execution is the same regardless of the value.

```
1  if( node.getLeft() != null && node.getRight() != null
       ) {
2    BSTNode minParent = node;
3    BSTNode min = (BSTNode)node.getRight();
4    while (min.getLeft() != null) {
5      minParent = min;
6      min = (BSTNode)min.getLeft();
7    }
8    node.setData(min.getData());
9    if (node == minParent)
10     minParent.setRight(min.getRight());
11   else
12     minParent.setLeft(min.getRight());
13 }
```

Program 7: Excerpts from the binary search tree delete routine

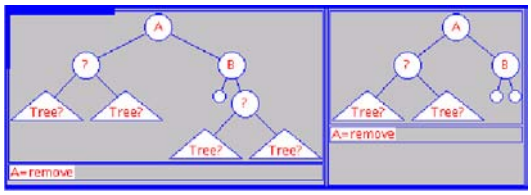

Figure 7: Two input data for the binary search tree leading to identical execution paths.

The same problem of extra branching in lazy initialization is present whenever branching does not depend on the initialized values. On the other hand, creating tests for such boundary cases (*i.e.* `null`s) might reveal some bugs that would otherwise be missed.

5. CONCLUSIONS

The work presents a novel idea of extracting test schemas and test data. Test schema is defined to be an abstract definition from where (several) test data can be derived. The reason for separating these two concepts is to provide automatic visualizations from automatically produced test data and therefore from what is tested.

On a concrete level, the work has concentrated on using the JPF software model checker in test data generation. Known approaches of using JPF in test data generation (*i.e.* concrete method sequence exploration, symbolic method sequence exploration, and generalized symbolic execution with lazy initialization) have been described. In addition, new approaches have also been developed:

Use of Comparable interface that removes the need of annotation in the previous symbolic test data generation approaches. The drawback of the approach is that only programs using comparables can be used.

Use of probes to remove the manual invariant construction needed by the lazy initialization.

Both new approaches are also a step from model based testing towards test creation based on real Java programs. Automatic assessment of programming exercises is not the

only domain where the results of this work can be applied. Other possibilities are for example:

- Tracing exercises is another educational domain where the presented techniques can be directly applied. In tracing exercises test data and algorithm are given for a student. The objective is to simulate (or trace) the execution (*e.g.* [14]). The problem of test adequacy (*i.e.* providing test data for students) is the same as addressed in this research.
- Traditional test data generation can also benefit from our results. We believe that the idea of hiding the symbolic execution behind the Comparable interface is interesting. Extra branching resulting from the Comparable construction is not that bad, because it is nearly the same as boundary value testing (*e.g.* [9]). Instead of creating one test data for a path with the constraint $a \leq b$, two tests are created: $a = b$ (*i.e.* the boundary value test) and $a < b$.

As a summary, interesting concepts and techniques to make automatic test data generation more attractive in teaching and especially automatic assessment are presented. Results can be reasonably well generalized and applied on other contexts than automatic assessment of programming exercises. However, the work is the first step to bring formally justified test data generation and education closer to each other.

Acknowledgements: This article is based on my masters thesis work and therefore I thank my thesis instructor Ari Korhonen and supervisor prof. Lauri Malmi.

6. REFERENCES

[1] K. Ala-Mutka, T. Uimonen, and H.-M. Järvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

[2] C. Artho, D. Drusinksy, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop*, volume 2589 of *LNCS*, pages 87–108. Springer-Verlag, 2003.

[3] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes. Validating use-cases with the AsmL test tool. In *Proceedings of 3rd International Conference on Quality Software*, pages 238–246. IEEE Computer Society, 2003.

[4] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, 1993.

[5] G. Brat, W. Visser, K. Havelund, and S. Park. Java PathFinder - second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification, Chicago, Illinois*, July 2000.

[6] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[7] D. Coward. Symbolic execution and testing. *Inf. Softw. Technol.*, 33(1):53–64, 1991.

[8] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[9] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[10] D. Jackson and M. Usher. Grading student programs using assyst. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, New York, NY, USA, 1997. ACM Press.

[11] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[12] S. Khursid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings 9th International Conference on Tools and Algorithms for Construction and Analysis*, volume 2619 of *LNCS*, pages 553–568. Springer-Verlag, April 2003.

[13] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] A. Korhonen, L. Malmi, and P. Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of the 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[15] A. Mitrovic and S. Ohlsson. Evaluation of a constraint-based tutor for a database language. *International Journal of Artificial Intelligence in Education*, 10:238–256, 1999.

[16] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of 11th International SPIN Workshop*, volume 2989 of *LNCS*, pages 164–181. Springer-Verlag, 2004.

[17] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.

[18] L. Salmela and J. Tarhio. ACE: Automated compiler exercises. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education*, pages 131–135, Joensuu, Finland, October 2004.

[19] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Proceedings of the sixth conference on Australian computing education*, pages 317–325. Australian Computer Society, Inc., 2004.

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng*, 10(2):203 – 232, April 2003.

[21] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM Press, 2004.

[22] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.

Automatic Tutoring Question Generation During Algorithm Simulation

Ville Karavirta*
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
vkaravir@cs.hut.fi

Ari Korhonen
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
archie@cs.hut.fi

ABSTRACT

High user interaction is the key in the educational effectiveness of algorithm visualization (AV). This paper introduces integration of an AV system with an interaction component in order to add support for the responding level of the user engagement taxonomy. We describe the resulting AV system, which is capable of providing algorithm simulation exercises with pop-up questions that guide the student in solving the exercise. This research aims at providing a system usable in teaching as well as in validating the engagement taxonomy.

Keywords

algorithm simulation exercises, automatic assessment, algorithm visualization, engagement taxonomy

1. INTRODUCTION

Algorithm visualization (for the rest of the paper, abbreviated as AV) gives a visual representation of an algorithm on a higher level of abstraction than the actual code. The aim of AV is to help humans to grasp the workings of the algorithm through visualization. Although AV has been used in education for more than two decades with varying results, the current belief is that AV must provide interaction between the animation and the student in order to be educationally effective [2].

Further research on the topic of *AV effectiveness* has come up with a taxonomy to measure the user engagement with the AV (system). This *engagement taxonomy* [13] defines six levels of engagement:

No viewing – no use of AV technology.

Viewing – passive viewing of AV material where the student only controls the visualization's execution, for example, by changing the speed and direction of the animation.

Responding – the student is engaged by asking questions about the AV material. These questions can be, for example, about the efficiency or prediction of the next step of the algorithm.

Changing – requires the student to modify the AV material. This can be done, for example, by changing the input data.

Constructing – the student is required to constructs his/her own AV material related to an algorithm.

Presenting – the student presents a visualization for an audience.

The hypotheses of the taxonomy are that there is no significant difference between levels no viewing and viewing, and that the higher the level of engagement the better the learning outcomes of students.

The above hypotheses have been studied over the years. Before the taxonomy was even defined, Jarc et al. conducted an experiment comparing the levels viewing and responding [4]. The results of the survey suggested, that the students using more interactive material scored better on difficult topics, but poorly in overall. None of the differences were statistically significant, though. Lauer reports on an evaluation comparing three different levels of engagement: viewing, changing, and constructing [8]. Neither this study found statistically significant differences, although the group using changing performed slightly worse on average. Grissom et al. experimented to compare levels no viewing, viewing, and responding [1]. The results show that learning improves as the level of student engagement increases. The difference between no viewing and responding was statistically significant. None of the studies mentioned above have compared, for example, level responding with changing, and constructing.

We have developed a growing set of algorithm simulation exercises that support engagement levels viewing, changing, and constructing. However, as claimed by Rößling and Häussage, the support for the responding level in the exercises in MatrixPro [6] and TRAKLA2 [9] is *"too limited to use for experiments that require this level of engagement"* [15]. Thus, to be able to use TRAKLA2 / MatrixPro in experimental studies requiring engagement in all the levels (except possibly presenting), we wanted to add proper support for responding level as well.

In this paper, we will describe an approach to extend our systems to support the responding level by adopting the AVInteraction software module [15] developed in the Technische Universität Darmstadt, Germany. The same software module is utilized by other AV systems as well. This will allow us to conduct and repeat experiments comparing the different levels of engagement and hopefully validating the hypotheses presented in the engagement taxonomy. We will also point out new research questions we would like to study.

The rest of this paper is organized as follows. Section 2 introduces related work done on this topic. Next, Section 3 describes the concept of algorithm simulation exercises.

*Corresponding author.

Section 4 in turn concentrates on our implementation of responding level interaction in algorithm simulation exercises. Finally, Sections 5 and 6 discuss the usefulness of our approach and concludes this papers main ideas, respectively.

2. RELATED WORK

Our aim is to support the engagement level responding in our existing system. However, as there already are systems supporting this level (see, e.g., ANIMAL [14] or JHAVÉ [12]) we do not want to reinvent the wheel by implementing yet another interaction piece of software, but rather reuse existing work developed by others.

A package intended for the kind of interaction we are looking for is AVInteraction [15]. AVInteraction is a tool-independent interaction component designed to be easily incorporated into Java based AV systems. The component offers a parser and a graphical interface for the interaction events. In addition, it includes an option to evaluate the answered questions. The current version of the component can handle different types of questions:

- *true /false* questions, where the user chooses from two alternatives
- *free-text* questions, where the user can type in free text. The answer is evaluated by string matching that allows several possible correct answers determined in the question specification.
- *multiple choice* questions, where the user selects a subset of the given choices.

The developers of the AVInteraction component have used it in their algorithm visualization system ANIMAL [14]. In ANIMAL, the user is asked questions about the animation while viewing it. Usually, these questions require the student to predict what will happen next in the animation.

The AVInteraction package has been successfully integrated to another system as well. Myller [11] introduced a version of Jeliot 3, a program visualization system for novice programmers [10], that supports automatic generation of prediction questions during program visualization. The approach taken also used the AVInteraction to bring up the questions.

3. TRAKLA2 EXERCISES

TRAKLA2 [9] is an automatic exercise system for learning data structures and algorithms. The system provides an environment to solve *algorithm simulation exercises* that can be automatically graded. In this environment, the students manipulate conceptual views of data structures simulating actions a real algorithm would perform. All manipulations are carried out in terms of graphical user interface operations. The system records the sequence of operations that is submitted to the server. The submitted sequence is compared with a sequence generated by a real implemented algorithm, and feedback is provided for the student. The feedback is based on the number of correct steps in the sequences of operations.

For example, a typical TRAKLA2 exercise is "insertion into a binary heap". In this exercise, the student simulates how a number of keys are inserted into an initially empty binary heap. An array of keys is provided and the task is to move them one-by-one into the tree representation of the heap. In addition, the heap order property must be restored after each step (heapify). The feedback is not provided until all the keys are inserted. Thus, there is a number of ways to solve the exercise incorrectly. This is especially useful in case the user has a misconception of how the algorithm works [16]. For example, in this exercise, the student might have difficulties to understand how recursion works. Thus, he/she might never do the recursive step in the heapify procedure (common error in final examination). This is why the feedback provided should be designed in such a way that it promotes reflective thinking, i.e., the feedback should reveal the misconceptions and change the way of thinking: a process we call learning.

The initial data for each exercise is random. For example, the array of keys in the previous example is randomized for each exercise instance. This allows the student to request grading for the solution an unlimited number of times. However, after each grading action, the student cannot continue solving the exercise with the same data. Instead, the exercise must be re-initialized with new random data.

In addition, the student can request the model solution for the exercise at any time. The solution is presented as an algorithm animation that the student can freely browse backwards and forwards. However, as with grading, the student has to reset the exercise and start with fresh random initial data before he/she can resubmit the solution again. Between two consecutive submissions, the student is encouraged to compare the submission with the model solution and find out the error made (i.e., reflect the possible misconception). Yet, the number of resubmissions can be unlimited.

The user interface of TRAKLA2 is a Java applet that is tailored to each exercise separately. The applet includes visualizations of data structures needed in the exercise, push buttons for requesting *reset*, *submit* and *model solution* for the exercise, as well as buttons for browsing one's own solution backwards and forwards. Simulation is carried out by drag-and-dropping data items (i.e., keys and nodes in data structures) or references (i.e., pointers) from one position to another. Some exercises also include additional push buttons to perform exercise specific operations such as rotations in trees. Thus, more complex exercises can be divided into several smaller tasks. For example, there are separate exercises for single and double rotations in AVL tree (both based on pointer manipulation). The final task, however, is to learn how to insert a number of keys into an AVL tree. This exercise has separate buttons to perform the corresponding rotations (instead of still doing them in terms of pointer manipulation).

Current selection of exercises deals with binary trees, heaps, sorting algorithms, various dictionaries, hashing methods, and graph algorithms. You can freely access the learning environment on the web, and create a test account to try out the exercises.[1]

3.1 Remark

Implementing new exercises is the hard part of the system.[2] However, it should be noted that any algorithm

[1] http://svg.cs.hut.fi/TRAKLA2/

[2] Currently, designing, implementing, and testing a new exercise takes roughly a week.

simulation exercise can be reused unlimited number of times with unlimited number of students.[3] Thus, after implementing an exercise, the human workload is independent of the number of students and submissions they do. This is due to the fact that the exercise is automatically assessed for each initialization, and each instance of the exercise is different from previous one. Thus, the students cannot copy the answers from each other. Actually, they need to think the solution anew for each submission. In addition, due to the model solution capability, the students can view any number of example solutions before, during, and after solving their own exercise.

4. IMPLEMENTATION

This section describes our implementation of adding the responding level to an existing AV system. Figure 1 shows an example of an exercise with the tutoring questions included. In the exercise, student is expected to color the nodes of a binary search tree to be a valid red black tree. The tutor gives guidance by asking questions whenever any of the red black tree properties are violated:

1. A node is either red or black.
2. The root is black.
3. Both children of a red node are black.
4. All paths from the root to any empty subtree contain the same number of black nodes.

The questions asked from students, however, are not restricted to these (explicit) properties. We can derive a number of (implicit) properties from these that we call rules. In the figure, for example, the question "How many children can a red node have?" is shown. The question was selected due to the fact that one of the nodes in the tree violates the implicit rule that says "a red node cannot have only one child." (but zero or two children). This rule follows from the properties.

PROOF. (Proof by Contradiction.) Assume to the contrary that a red node can have exactly one child. This child must be black by the property 3. Well then, there exist two paths that differ in their black lengths: the one that ends at the other side of the red node, and the longer path that ends after the black child. This violates the property 4, contradicting our assumption that a red node can have only one child. □

We have identified three different forms of questions that can be asked from the student:

tutoring questions – These are questions such as in Figure 1, which are asked while something is wrong within the student's solution. The most simple example of such question is to ask about the correct order of the alphabets if the student has made a mistake with them. The overall aim of this form of questions is to guide the student into the right direction in solving the simulation exercise.

detailed questions – These questions are not directly related to the simulation exercise, but rather ask some extra properties of the data structures manipulated in the exercise. Examples of such questions are (in the binary search tree exercises) questions about the height of the trees, and how many nodes fit in a tree if the height is limited. The aim of this form of questions is to increase the student's knowledge about the topics related to the exercise.

thinking questions – These are kind of mixed tutoring and detailed questions in which the same question might help the student to continue with the simulation, but at the same time, possibly increase the knowledge about the topic. In the red black tree exercise, for example, some students might try to prove the new implicit rule emerging from the question based on the given properties. The proofs are typically simple, as demonstrated above, but difficult for novices. Thinking questions can be promoted by asking them more explicitly, for example, in class room sessions. However, visual algorithm simulation exercises and related tutoring questions can motivate the student to actually think of this proof due to the fact that the "proof can be seen" (in Figure 1, spot the erroneous red node that has only one child, and read the proof again).

There are major differences in the nature of the questions and approaches between our tutor and the other systems using the same AVInteraction component. In ANIMAL the questions are loaded once the animation is loaded and are in a way static questions presented at predefined time. The engagement is mainly a combination of levels viewing and responding. Our approach combines engagement levels responding and changing/constructing[4]. Furthermore, both ANIMAL and Jeliot 3 require the user to respond to the questions[5], where in our implementation, the responding to the questions is voluntary.

Another difference is that ANIMAL and Jeliot 3 are usually used to ask questions about the next step of the animation, so to *predict* the result. Our approach is to use the questions to *guide* or *tutor* a student to construct a correct simulation of the algorithm in question. Actually, the response is not that important at all. That is to say, we know that the students know the correct order of alphabets in the simple tutoring questions described above. Thus, in our approach, the most crucial thing is the thinking process triggered by a question.

4.1 Technical Aspects

We used the AVInteraction component to add support for the responding level of the engagement taxonomy to our algorithm simulation exercises. As the exercises currently assume that the student always has the control, we included the questions in a way where the student is not forced to answer them. Due to this, we do not use the responses for grading purposes, but merely to give feedback and guide the process of solving an exercise.

The integration of the AVInteraction component to the exercises was straightforward and simple. Our approach,

[3]There are some 40 exercises implemented that already cover most common data structures and algorithms.

[4]The level of engagement depends on the nature of the exercise. For a more detailed discussion, see [7].

[5]In Jeliot 3, the user can turn on/off the questions but cannot leave a shown question unanswered.

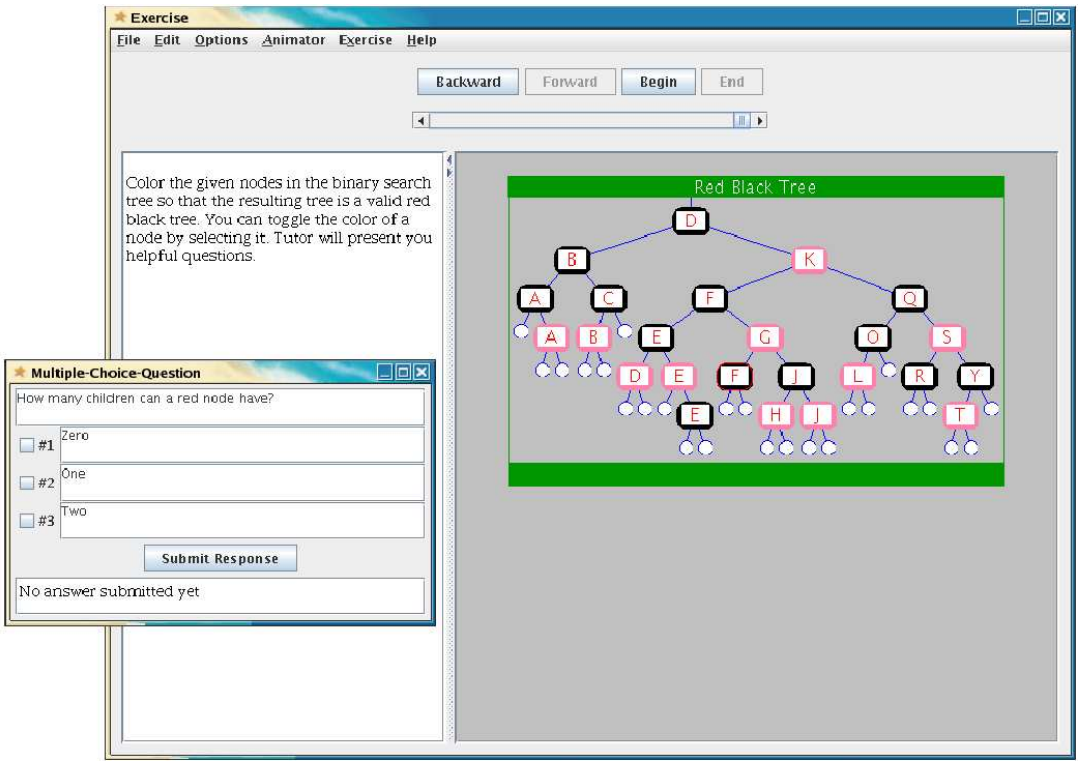

Figure 1: An example algorithm simulation exercise with tutoring questions. The question "How many children can a red node have?" is asked because the red node E has only one child node, which is against the red black tree properties.

however, required dynamic generation of questions (the questions depend on the exercise instance that is randomized). Thus, some problems were caused by the design of the component in which all the questions were loaded upon initialization.

The AVInteraction package did not fill all the needs we had once starting the implementation. There is room for improvement in the use of visualizations with the questions. For example, it would be good if one could add visualizations as part of the questions. Furthermore, a feature that allows students to answer the questions by manipulating the visualizations in the exercise instead of through the AVInteraction GUI would be useful. In a way, this has already been implemented in our version. For example, in the red black tree coloring exercise, the question "What color is the root node of the red black tree" can be "answered" by coloring the root node black, which causes the question to be changed.

One important aspect of introducing such tutor to more exercises is the amount of work required to add the responding functionality. The introduction of the questions does require some effort in addition to the implementation of a new exercise. However, this effort is mostly needed to design a sensible set of questions and the logic used to determine when and why to ask them. Adding the questions to the exercises only requires implementation of a Java interface with one method.

5. DISCUSSION

The aim of this project was to extend the TRAKLA2 system to support more of the levels of engagement taxonomy. We first review the levels and give examples of the use cases possible with this extended version. It should be noted that these are merely examples, and it is possible to design simulation exercises for our system where some certain levels are supported. For example, an exercise with support only for responding level.

Viewing is allowed in several phases during the simulation process. First, the concept of visual algorithm simulation is based on the visualizations of data structures that the user is supposed not only to view, but also to modify during the process. In addition, in the algorithm simulation exercises, TRAKLA2 provides model solutions in terms of algorithm animations that also correspond to the viewing level.

Responding is fully supported after integrating AVInteraction module to be accompanied with the exercises. The major differences between our approach and some previous experiments are the following:

1. Our system allows *dynamic questions* to be asked that are related to the current state of the simulation exercise. This is different, for example, compared with ANIMAL in which the questions are static and loaded together with the animation. Thus, in ANIMAL, the questions are predefined and always the same from one run to another. Our approach, however, allows that the question is parametrized and depends on the current run.
2. Our system has *non-blocking questions* during the animation / simulation. For example, in Jeliot 3, it is possible to ask dynamic questions, but the student cannot continue the first task (animation) until the question is answered. Our approach would allow both, blocking and non-blocking questions, but

we have decided to use non-blocking questions since the focus is on the simulation that is graded. Thus, the animation / simulation can be continued, and the question might even be answered by doing the correct step in the simulation. This is an especially useful feature if the question is intended to guide the student more than address some detail related to the topic.

3. Algorithm simulation exercises intrinsically have the property that the student is required "to predict" the next step in the simulation process. In ANIMAL and Jeliot 3, this is something that is promoted through responding: the student is asked an explicit question "predict what happens next". Thus, in our case, we consider the "responding level" to be something that is already incorporated in the "changing" and "constructing". Of course, this new implementation makes responding and this kind of predicting more explicit.

Changing entails modifying the visualization. In general, visual algorithm simulation supports this level very well. For example, in MatrixPro, the user can freely choose any input for the algorithm under study in order to explore the algorithm's behavior in different cases. Such free experimentations could be connected to some open ended questions, but the TRAKLA2 exercises are more closed in this respect. Thus, in the current set of algorithm simulation exercises, this level is not promoted very well. However, the form of engagement is more on the next level.

Constructing requires the students to construct their own visualizations of the algorithms under study. In terms of visual algorithm simulation, this means that students simulate a given algorithm step-by-step by manipulating visual representations of data structures. In TRAKLA2, this process is supported by a Java applet that automatically updates the representations according to the modifications. Thus, the student can concentrate on the logic and behavior of the algorithm, i.e., content, and ignore the hard part of drawing complex illustrations. The students construct visualizations, but do not draw them, which means that the focus is on the process, not on a single outcome.

Presenting entails showing an algorithm visualization to the audience for feedback and discussion. In general, visual algorithm simulation is an effortless way to produce such visualizations [3], but our experiences are limited to lecture situations where it is the lecturer that gives the presentation. However, MatrixPro could be used in this engagement level, for instance, in a setup that requires the student to prepare the visualization on-the-fly. For example, the students could be told which algorithms and data structures to get familiar with, but the actual visualization must be constructed during the presentation (they can learn to simulate the algorithm with any input they wish, but do not know which input they must use in the final presentation).

Some tools are difficult to place in the engagement taxonomy. This might be due to the fact that the levels are not orthogonal (the semantic distance among the levels could be higher if we choose some other levels instead of these four; omitting "no viewing", and recalling that the 1st level, viewing, was considered to be included in all the other four in the first place). We back up our claim by arguing that 1) it is not always clear whether some activity belongs to a certain engagement level or perhaps on many levels at the same time; and 2) it is not always clear how wide is the scope of a certain level. First, in algorithm simulation exercises, the student is constantly supposed to "predict the next step" something typically considered to be on the **responding level**. Still, while doing these exercises the students are clearly **changing** the data structures involved as well as **constructing** the animation sequence (i.e., algorithm visualization) submitted as an answer. Second, if we ask questions about the AV material, it is not clear whether this is included in the changing and constructing levels. For example, our approach to integrate AVInteraction is such that the student can "answer" the question (e.g., what's the color of the root node in a red black tree?) explicitly by choosing the correct alternative, or by **changing** the visualization while **constructing** the next step (e.g., toggling the color of the root node). If the student picks up this latter alternative, is this action considered to belong in the responding, changing, or constructing level?

Another interesting measure to consider is whether or not the inclusion of the responding level in the algorithm simulation exercises lowers the number of submissions students use. In a previous study, we have found out that there exists a group of students who use resubmissions carelessly [5]. Thus, our hypothesis is that including the responding level in terms of tutoring questions could improve the performance of these students.

6. CONCLUSIONS

In this paper, we have described work in progress on the next generation of the visual algorithm simulation (exercises) that support also interactive questions, i.e., responding level in the engagement taxonomy by Naps et al. [13]. While the prototype implementation has a number of enhancements, we focus on discussing the meaning and importance of such new functionality in our context that has certain differences compared with other AV systems. This has also an impact on how to interpret the levels of the engagement taxonomy, especially while planning research setups to test the hypotheses set by the authors of the engagement taxonomy (e.g., whether an effective instructional AV must allow the visualization designer to ask questions of the student).

The aim was to support the responding level also in the context of algorithm simulation exercises. The purpose of such questions typically include the idea of predicting the next step in an algorithm animation. However, as the algorithm simulation exercises intrinsically include the idea of "predicting the next step", we wanted to allow more options to ask questions. First, the questions do not need to block the simulation process, but the student can answer the question either explicitly (by providing the correct answer) or implicitly by continuing the simulation until the question is fulfilled by other means. This requires that the questions are dynamic in such a way that they are dependent on the current state of the exercise (i.e., not only the input of the algorithm has an effect to the question and its answer(s), but also the state of the simulation). This is different compared with algorithm animations that have only one possible execution path after the input is set. In algorithm simulation, there are many execution paths (even though only one of them is considered correct) as the exercises should also allow the students to make errors. Second, the scope of the questions do not

need to be limited to predicting what happens next, but they can also guide the students to broaden their knowledge about the topic. Such questions can include tutoring questions (to guide how to proceed with the simulation) and detailed questions (to focus on certain properties of the data structures and algorithms) as well as some mixture of these two. Finally, the responding level questions can easily be combined with other engagement levels. Our hypothesis is, however, that it might be difficult to compare certain levels of engagement directly (i.e., responding with constructing). Yet, this work makes it explicit that the responding level exists if required. This should be taken into account while planning new research setups. For example, if the idea is to compare responding with changing, it might turn out that it is intrinsically impossible to have a setup in which changing does not include any kind of responding (unless changing is forced to be implemented in an unnatural way). This point of view may also give a valuable insight into previous studies that have come up with mixed results.

Acknowledgments

We would like to thank Tomi Lehto for making the implementation work for this research.

This work was partially supported by the Academy of Finland under grant number 210947.

7. REFERENCES

[1] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in CS education: comparing levels of student engagement. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 87–94, New York, NY, USA, 2003. ACM Press.

[2] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.

[3] P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander. Taxonomy of effortless creation of algorithm visualizations. In *Proceedings of the 2005 international workshop on Computing education research*, pages 123–133, New York, NY, USA, 2005. ACM Press.

[4] D. J. Jarc, M. B. Feldman, and R. S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. In *The proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 377–381, Austin, Texas, 2000. ACM Press, New York.

[5] V. Karavirta, A. Korhonen, and L. Malmi. On the use of resubmissions in automatic assessment systems. *Computer Science Education*, 16(3):229 – 240, September 2006.

[6] V. Karavirta, A. Korhonen, L. Malmi, and K. Stålnacke. MatrixPro - A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33, The University of Warwick, UK, July 2004.

[7] A. Korhonen and L. Malmi. Taxonomy of visual algorithm simulation exercises. In *Proceedings of the Third Program Visualization Workshop*, pages 118–125, The University of Warwick, UK, July 2004.

[8] T. Lauer. Learner interaction with algorithm visualizations: viewing vs. changing vs. constructing. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 202–206, New York, NY, USA, 2006. ACM Press.

[9] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267 – 288, 2004.

[10] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 373 – 376, Gallipoli (Lecce), Italy, May 2004.

[11] N. Myller. Automatic prediction question generation during program visualization. In *Proceedings of the Fourth Program Visualization Workshop*, 2006. To appear.

[12] T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the SIGCSE Session*, pages 109–113, Austin, Texas, Mar. 2000. ACM Press, New York.

[13] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

[14] G. Rößling and B. Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

[15] G. Rößling and G. Häussage. Towards tool-independent interaction support. In *Proceedings of the Third Program Visualization Workshop*, pages 110–117, The University of Warwick, UK, July 2004.

[16] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student misconceptions – a case study of the build-heap algorithm. *Computer Science Education*, 16(3):241–255, September 2006.

Do students *SQLify*?
Improving Learning Outcomes with Peer Review and Enhanced Computer Assisted Assessment of Querying Skills

Michael de Raadt
Dept. Mathematics and Computing
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
deraadt@usq.edu.au

Stijn Dekeyser
Dept. Mathematics and Computing
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
dekeyser@usq.edu.au

Tien Yu Lee
Dept. Mathematics and Computing
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
leet@usq.edu.au

ABSTRACT
In recent years a small number of web-based tools have been proposed to help students learn to write SQL query statements and also to assess students' SQL writing skills. *SQLify* is a new SQL teaching and assessment tool that extends the current state-of-the-art by incorporating peer review and enhanced automatic assessment based on database theory to produce more comprehensive feedback to students. *SQLify* is intended to yield a richer learning experience for students and reduce marking load for instructors. In this paper *SQLify* is compared with existing tools and important new features are demonstrated.

Keywords
Peer Review, Computer Assisted Assessment, Web-based Learning, Databases, SQL.

1. INTRODUCTION
SQL is the dominant language for defining and manipulating databases. SQL querying skills are highly valued in the computing industry and as such teaching of SQL in tertiary institutions rivals the importance of programming instruction.

Teaching students to write SQL queries has always been an onerous task for both instructors and students. Students suffer a number of identified difficulties in learning SQL (shown in section 1.1). To assist students in overcoming these difficulties several tools have been suggested which provide a simple environment for students to write and test queries against databases, receive immediate feedback which is more informative than what can be offered by a Database Management System (section 1.2).

For instructors, marking queries on paper can be tedious and error prone. Integrating tutoring systems into assessment systems can allow instructors to mark the products of student learning in a more efficient and accurate manner.

Current systems for tutoring and assessment have proven their worth. A new system, *SQLify*, described here, combines most features present in existing systems.

- Visualization of database schema
- Visualization of query processing
- Feedback on query semantics
- Query assessment (using heuristics)
- Consistent grading between students and markers
- Relational Algebra expressions support

No single system other than *SQLify* combines all the features above. *SQLify* also incorporates several important new features to further improve learning outcomes for students and assist instructors.

- Query assessment (using CQ query equivalence)
- Scoring correctness beyond binary correct/incorrect
- Use of peer review for assessment

1.1 Difficulties in Learning SQL
SQL has a simple syntax with a limited set of commands, yet it is possible to create complex queries with powerful results.

Even as early as 1978 Shneiderman [16] describes difficulties encountered by students. Shneiderman's study showed students can produce queries equally well in natural language and in, at that time, SEQUEL, but produced many more errors before achieving a correct artificial query.

Sadiq *et al.* [14] suggest the "straight forward syntax of the SQL SELECT command is often misleading, and generates an impression of simplicity in learners' minds. Sadiq goes on to compare the declarative nature of programming languages, which require users to think in steps, with SQL, where users think in sets which can be difficult for learners.

Mitrovic [11] suggests learners struggle with the burden of having to memorize database schema and produce incorrect solutions because of this. Mitrovic also reports difficulty with grouping, join conditions and the difference between universal and existential quantifiers. These difficulties are also suggested by Kearns *et al.* [9].

1.2 SQL Tutoring and Assessment Systems
In efforts to overcome identified problems associated with learning SQL, a number of tools have been created at various institutions, each allowing practice with feedback beyond that of a normal DBMS. Additional interactive feedback is used to overcome semantic misunderstandings and oversimplifications. Visualization of schema and query processessing is used to overcome memorization problems. Some SQL teaching tools also offer integration with assessment in undergraduate courses. A number of such tools are described in literature.

- *SQLator*, a tool created by the University of Queensland in 2004 and used extensively at the time [14].
- *AsseSQL*, a tool created by the University of Technology, Sydney also in 2004 [12, 13].

- *SQL-Tutor*, described in [11] and developed at the University of Canterbury in Christchurch in 1998. This system attempts to provide intelligent feedback on students' attempts to create SQL queries.
- *eSQL*, proposed in 1997 to help visualize the process of query processing [9].
- *RBDI* is a command line tool allowing students to practice their query writing skills in SQL, relational algebra and relational calculus. An extension of this, *WinRBDI*, is described in literature [8].

All systems are used to teach students to write SQL statements. *SQLator* and *AsseSQL* are used to assess student queries and will be the focus of the remaining review.

Prior and Lister [13] present *AsseSQL* as an online tool which allows entry and execution of SQL queries by students. They suggest an electronic interface creates a more authentic task than writing queries on paper and may encourage a deeper learning approach. Students are allowed access to *AsseSQL* for practice, but the ultimate use of this system appears to be in closed examinations under supervised, time constrained conditions within a computer laboratory. As well as being given a problem to solve, students are shown the desired result of the query they are to write as part of the problem description; this is justified as an attempt to overcome students' poor English skills. Apart from being online, these conditions and aids appear to create an unauthentic setting for student learning. Professional database users do write queries with computers, but not in these conditions. They will not know the results of a query before they create it. The system provides immediate feedback, but this is limited to the correctness of the solution provided by the student. No comments or suggestions for improvement are provided. While this reduces the marking load on instructors, it does not correct students' misunderstandings or encourage further learning. Three forms of evaluation on *AsseSQL* are provided: results of a student attitudinal survey, evidence of a focus group and opinions of instructors. These evaluations show that an online SQL assessment tool is worth pursuing; however no validation of the system against student outcomes or results is suggested.

A clearer validation is presented by Sadiq *et al* [14] for *SQLator* which showed student engagement through voluntary student practice statistics and improved results in final grades. The *SQLator* system attempts to judge the correctness of submitted queries and also provide intelligent feedback to "enhance [student's] learning experience." It is not clear how student results are used for assessment purposes or how students are motivated to use *SQLator*.

The papers describing the above mentioned tools focus on the resulting improvements in educational outcomes. None of these papers describe in detail the inner workings of their system and show minor regard to relational database theory. There is little mention of SQL teaching tools outside computing education.

Both *AsseSQL* and *SQLator* apply only a simple binary grading to queries submitted by students. While the creators of *AsseSQL* argue for the sufficiency of this *right-or-wrong* approach, a greater objective discrimination of quality is possible using a more sophisticated grading system (see Section 2).

Both *AsseSQL* and *SQLator* use heuristic methods to evaluate queries entered by students. This involves running the submitted query on a test database, and comparing the output with that of the query included in the definition of the problem. It is possible for students to cheat by creating simple queries that produce the desired output for the given database instance, which cannot be generalized to all instances of the database. For example, assume a student was asked to write a query to *obtain names of employee who work in IT Department*. With two tables, an employee table and a department table, normally a join would be required to discover the department ID of the IT Department and then discover which employees are in that Department.

```
SELECT name FROM emp, dept WHERE
emp.deptno=dept.deptno AND
dept.deptname='IT';
```

A student, seeing an instance of the database and knowing that the `deptno` for the IT Department is `5`, could cheat by writing a query which produces the correct output without consulting the `dept` table.

```
SELECT name FROM emp WHERE deptno=5;
```

Sadiq *et al*. [14] suggest *SQLator*, using heuristic comparison, marks a query as correct in 95% of relatively easy test cases. The success of the heuristic depends in part on the database instance used in the test; a badly designed instance reduces the level of correctness of this method. In [13], Prior and Lister propose extending *AsseSQL* to run an additional test on a second database not shown to students. While this may increase the correctness of evaluation, it is still only another heuristic test.

In database theory it is well known that queries in the class of Conjunctive Queries (CQ) possess an important property: it is decidable whether two queries are equivalent. The CQ class is a significant subset of SQL excluding the set operators (union, difference, intersection) and grouping statements. In the introductory *Database Systems* course at the University of Southern Queensland, more than 70% of the time spent on SQL is reserved for such queries. For this class of queries a computer assisted assessment tool should be able to evaluate correctness of submitted queries with 100% accuracy by examining the submitted queries alone. For queries that are not in CQ, a heuristic approach can still be used by comparing the output instance of the submitted query with that produced by the instructor's set solution query. Such queries can then be flagged for instructor moderation.

Some practical considerations regarding database systems are also unaddressed in the existing literature. The use of the DISTINCT keyword or *sorting* in a query makes it impractical to test equivalence using only the heuristic described above. Furthermore, both *AsseSQL* and *SQLator* seem vulnerable to SQL injection attacks. These include attempts to make unauthorised modifications to a database by taking advantage of the level of access provided by the interface. Care must be taken to check or rewrite a submitted query before it is evaluated by the database server.

The techniques used in automated SQL teaching and assessment tools can be readily used for relational algebra as well. This requires only a user-friendly (and, desirably, pedagogically sound) interface for entering relational algebra statements, and additional logic to convert students' algebra expressions into SQL. This conversion process is a well-documented procedure. This is not used in *SQLator* or *AsseSQL* but is partially achieved in *RBDI*.

With an automated assessment system it is possible to involve students in the assessment process using *peer review*. Accord-

ing to Saunders [15] peer learning is advantageous as "it offers the opportunity for students to teach and learn from each other, providing a learning experience that is qualitatively different from the usual teacher-student interactions". Peer review can be conducted in a number of ways. The form used with *SQLify* takes a student's submission and allows it to be reviewed by a number of student-peers, a process automated by the system and overseen by an instructor. Peer review allows students to evaluate the work of others which requires higher order thinking skills [1] through evaluating the work of peers and reflecting on their own work. With peer review, students also receive feedback from more than one source enriching the learning experience for students. Receiving feedback from peers can encourage a community of learning [2] which can in turn further encourage higher order thinking. Peer review involves students in the assessment process, encouraging increased engagement in the course and ultimately improved learning outcomes [4]. Peer review has been successfully incorporated in the assessment of student work in various fields, including computing [3, 4, 5, 10] with demonstrated improvements in students' learning outcomes. Peer review, when used as an assessment tool, can also reduce the assessment workload of instructors. Both *AsseSQL* and *SQLator* create only a single channel of communication between the student and the instructor via the system. No other forms of communication (eg., peer to peer) are mentioned as being part of these systems or used along-side these systems.

In the next section, the *SQLify* system is proposed. The following section shows examples of a hypothetical implementation of *SQLify*. In the final section, conclusions and possible future extensions of the system are suggested.

2. THE *SQLify* PROPOSAL

Having compared and evaluated existing computer assisted learning and assessment tools, we now turn to the description of *SQLify* (pronounced as *squalify*) which aims to improve on existing solutions on several different fronts. Specifically, the following requirements have driven the design of *SQLify*:

- Provide rich feedback to students in an automated and semi-automated fashion;
- Employ peer-review to enhance learning outcomes for students (through students conducting evaluations and receiving feedback from more sources);
- Use database theory combined with peer review effectively to yield a wider range of final marks;
- Judge the accuracy of reviews performed by students;
- Reduce the number of necessary moderations conducted by instructors, freeing them for other forms of teaching.

Hence, the main focus of *SQLify* is computer assisted practice and assessment using a sophisticated automatic grading system in combination with peer review.

The current implemention of *SQLify*, with a demonstration of available functionality is viewable from the project website [6].

2.1 Use of *SQLify*

The *SQLify* system is intended to assess a student's query writing skills through an online interface in the context of assignments and preparing for assignments. Student use of the system can be seen to fall into a series of phases.

1. Trial and submission
2. Reviewing peers' submissions
3. Receiving feedback and marks

As show in Figure 1 a student will submit solutions to a number of problems. The value of their submission will be judged by peers, the *SQLify* system and ultimately by the instructor.

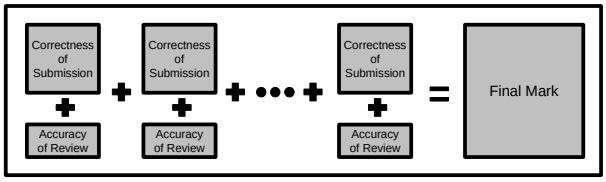

Figure 1: Components of Student's Mark

Students complete reviews of (usually two) other students submissions for which they are awarded marks. The accuracy of their submission determines the mark they receive for reviewing.

Finally the marks they received for submission and the accuracy of their reviews is summed for each question to form a final mark.

The following subsections describe in detail these three phases.

2.1.1 Trial and Submission

Students are able to develop and trial their query answers to a specific set of problems using *SQLify* and immediately see how the automatic grading system evaluates their work. The *SQLify* system will give one of (a limited set of) the levels of correctness shown in Table 2. Students may trial their solutions indefinitely without submitting their query answers. The mark they are shown during this trial period is not necessarily what they will receive from the instructor for the correctness of their submission; this is given later by the instructor under advisement of the student's peers and the *SQLify* system. When the student is happy with their work they may proceed to submitting query answers to assignment problems.

Students completing assignments using *SQLify* will typically be given a number of English-language problems (say three to

Table 1: Comparison of existing tools and SQLify

Feature	*SQLator*	*AsseSQL*	*SQL-Tutor*	*eSQL*	*WinRBDI*	***SQLify***
Modelling of student to individualize instructional sessions			✓			
Visualization of database schema			✓	✓		✓
Visualization of query processing						✓
Feedback on query semantics			✓		✓	✓[a]
Automatic assessment (using heuristics)	✓	✓[b]				✓[c]
Automatic assessment (using CQ query equivalence)						✓
Use of peer review for assessment						✓
Relational Algebra expressions support					✓	✓[d]
Special treatment of DISTINCT and ORDER BY						✓
SQL-injection attack countermeasures						✓

✓[a] in practice mode only
✓[b] on two instances (proposal only) ✓[c] for queries not in CQ
✓[d] currently being implemented

Table 2: Levels implied by evaluation sentences. Different sentences may by used by reviewing students, the *SQLify* system, and the instructor. Internal assessment values (last column) are *possible* values for each level which may be set by the instructor.

Level	Description	Student can use	System can use	Instructor can use	Possible internal value for query
L0	Syntax, output schema, and query semantics are incorrect	✓	✓	✓	0%
L1	Syntax is correct, schema and semantics incorrect	✓	✓	✓	20%
L2	Syntax and schema correct, semantics are incorrect	✓	✓	✓	30%
L3	Syntax and schema correct, semantics are largely incorrect			✓	40%
L4	Syntax and schema correct, semantics seem largely incorrect (not sure)	✓			70%
L5	Syntax and schema correct, semantics are just adequate			✓	80%
L6	Syntax and schema correct, semantics seem largely correct (not sure)	✓	✓		90%
L7	Syntax, schema, and semantics are correct	✓	✓	✓	100%

five) that he or she would translate to SQL or Relational Algebra. The problems are well defined descriptions of authentic, real world problems. Students' query answers are submitted through a web form shown in Figure 2 which demonstrates a simple query description, the database schema, links to a visualization of an instance of the database and to an output schema, and a text area where the student can enter their query answer. The student can also be supplied with hints and comments, and also with the desired output schema for the query (not the desired output instance), if so determined by the creator of the problem.

To evaluate relational algebra expressions students use an interface that helps construct syntactically correct algebra expressions. An algorithm translates the submitted algebra expression to an equivalent SQL statement. The generated statement is then processed in the same way as a normal SQL statement.

Once a query is submitted to the system it is checked for SQL injection attacks. First, tables referenced in the FROM clause of the submitted statement need to appear in the source database schema, or the query will be rejected. Second, the WHERE clause is analyzed and possibly rewritten using mainstream SQL injection countermeasures.

Students are not notified if their submitted queries are syntactically incorrect (although they should have been able to determine this themselves by trialing their submission).

Students receive feedback about their submission in the final phase (see section 2.1.3).

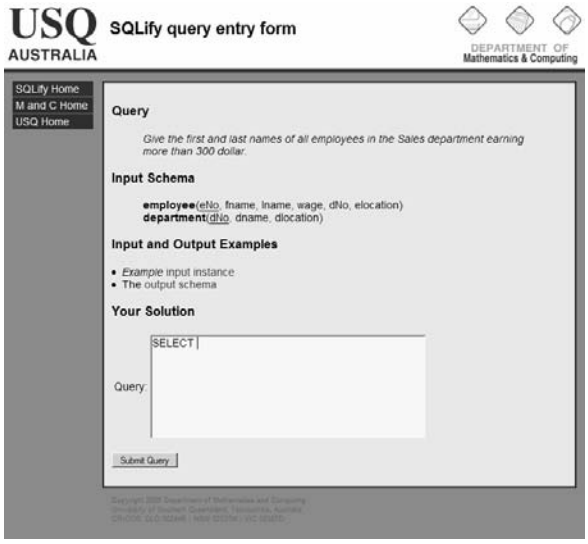

Figure 2: The form for query input

2.1.2 Reviewing Peers' Submissions

SQLify is used with a pre-existing peer review system defined in [4] and integrated with *SQLify* as follows.

After submitting, most students will be able to immediately proceed to complete reviews allocated to them. A small pool of early-submitting students (usually four) will wait until enough submissions have accumulated before they can proceed to reviews.

This *single step* submit-review process has been successfully applied [4] and has several advantages over a *two step* process (submit before deadline, review after first deadline and before a second deadline):

- only one deadline is needed,
- the majority of students are not required to return to the site for the sole purpose of completing reviews,
- students review the task they have just completed,
- students receive feedback from peers sooner, and
- students can work ahead in the course.

The system must facilitate reviewing in a way that maintains anonymity. The disadvantage of a *single phase* review allocation system arises when students can predict who they will review, in which case collusion between students is possible. This can be countered by complicating the review allocation process and keeping its workings secret, by requiring each submission to be reviewed by more than one peer and by comparing the accuracy of a student's review to a final correctness mark.

When the system has allocated reviews to a student, reviewing can commence. The student is presented with a similar screen to what they used to input their query answer during the initial submission phase, but where they were previously able to enter their answer the system now shows a read-only query given by a peer. The reviewing student additionally sees the result of applying the query on the relevant database instance. The reviewing student then selects a level described by a sentence from the list shown in Table 2 that best describes their assessment of the correctness of the query answer. The list of possible levels given in Table 2 shows all available levels of which the reviewing student may choose levels marked with a tick in

the column titled "Student can use". No corresponding internal values are shown to the reviewing student. Reviewing students may express uncertainty by choosing a sentence that includes "I am not sure". This allows the system to assign a wider range of marks to reviews, but is also used to flag potential problems that need to be moderated by an instructor.

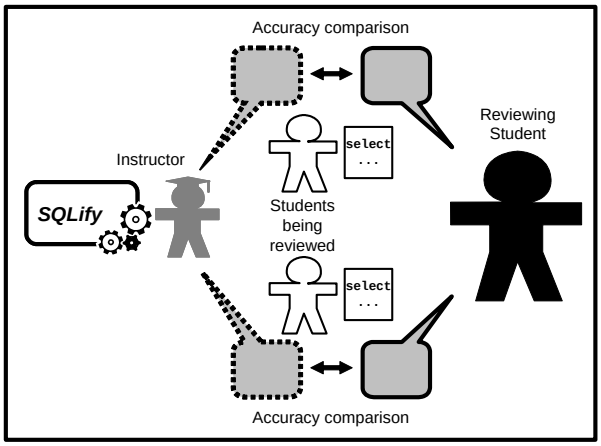

Figure 3: Checking Student's Peer Review Accuracy

By linking automatic assessment of queries with reviews given by students, it is not only possible to evaluate the correctness of queries, but also the accuracy of reviewers in judging that query. Students will review the work of two peers knowing that the reviews they perform will also be assessed as shown in Figure 3.

A student's review accuracy should be marked high when the level they selected for a peer's query answer is very similar to the level ultimately determined for that query answer by the instructor. Conversely, accuracy should be marked low when it differs greatly from the instructor's *correctness mark*. Hence, the formula for marking accuracy of a review performed by a student is quite simple.

$$accuracyMark = 100 - | correctnessMark - studentMark |$$

In other words, the mark given to a reviewer for the accuracy of their review depends on the difference to the correctness mark assigned by instructor. Note that this formula has the additional effect that when a student has signaled uncertainty (by picking level L4 or L6) they will not be awarded full marks for this review.

Giving fellow students a false high or low level evaluation which differs for the mark applied by an instructor will lose marks for the reviewing student.

As well as judging correctness levels for their peer's query answers, reviewing students are also required to leave a comment. Students are encouraged to give comments of praise or positive suggestions for improvement. This is arguably the most valuable part of the reviewing process for both the reviewer and the reviewee.

For the reviewer, peer reviewing is an opportunity to evaluate the work of a peer and in doing so, reflect on their own work. This requires higher order thinking skills [1] which will hopefully encourage greater learning outcomes.

For the reviewee receiving peer feedback means they will receive feedback from more sources than just the instructor or the system (see Figure 4). The information contained in comments can encourage a more personal relationship among students (even anonymously) and between instructors and students [4].

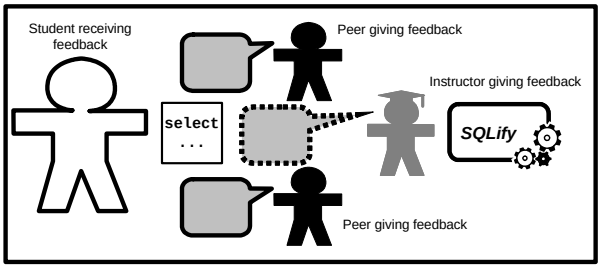

Figure 4: Feedback received by the student

For instructors, adding a comment allows elaboration on why a student may have lost marks and positive encouragement on their progress. The instructor may draw on a list of previously created comments to speed up the moderation process. This also provides consistency when multiple instructors are performing moderations.

It is important that students sense the instructor's involvement in the assessment process. They see the instructor as an authority and feel they deserve the attention of the instructor during the assessment process. It is possible for good students who produce excellent work, to be assessed equally by peers and the *SQLify* system. In such cases the instructor may elect to assign a mark based on the agreed standard of the work without performing moderation. If a student achieves this consistently through the semester, they may miss the instructors input in their assessment; they may then feel cheated by the assessment approach. It is possible to track how many times a student has been moderated by an instructor and set target levels of moderation at various points through the teaching period. This way each student can be satisfied with the attention they are receiving while still reducing the marking load on instructors.

Another potential of such a system is to allow students to flag peer reviews they believe to be incorrect for instructor intervention. Although quite often the instructor would be moderating such cases, this feature allows the student to express unhappiness with a review. This can remove some anxiety related to having their work assessed, in part, by peers.

2.1.3 Receiving Feedback and Marks

When all reviews of a student's work are complete, the instructor allocates a mark for the student's work based on the levels suggested by peers and by the *SQLify* system. Instructors must attend to submissions that have been assessed differently by each peer or by the system. Past experience [5] has shown that in at least half of normal submissions, peers alone are able to achieve non-conflicting reviews, so this means moderation is most likely to be unnecessary. In most cases the system can determine a level for a solution with absolute certainty so this further eases the marking load of the instructor.

One of the clearest benefits of using a single-step peer review system it that students receive feedback about their submission as soon as a peer has completed their review. Compared with a normal instructor marked assignment where students must wait until after the assignment deadline for feedback, previous use of the approach suggested here returns feedback to students within hours [5].

Once the peer review process is completed and the instructor has assigned marks to students the *SQLify* system can calculate a final mark for each student.

The system suggests a final mark for a student's assignment. It does so by summing both the correctness marks for each query answer and accuracy marks for the reviews conducted by that student. The weighting of correctness and review accuracy for each problem in each assignment could be varied according to the effort for each. An example would be weighting the correctness marks to 70% of the entire assessment and review accuracy marks to 30%. The instructor then chooses to accept or modify the suggested mark. Such marks may be released individually by the instructor or *en masse*. Details of how an accuracy mark is determined by the system and how an instructor determines their accuracy mark are given in [7].

3. EXAMPLE RUN THROUGH OF *SQLify* SYSTEM

To illustrate the workings of *SQLify*, two query problems are presented together with a description of how they would be evaluated.

The problems make use of a database with the following schema.

employee(eNo, fname, lname, wage, dNo, elocation)
department(dNo, dname, dlocation)

3.1 Problem Example 1

The first query problem (QP1) is an example of a Conjunctive Query (a problem in class CQ). In this class it is possible to conclusively determine if a supplied query is correct without employing heuristic comparison.

> *Give the first and last names of all employees in the Sales department earning more than 300 dollars (QP1)*

The instructor supplies a solution query that will be used by the system to test queries submitted by students.

```
SELECT fname, lname FROM employee E,
department D WHERE E.dNo = D.dNo AND
dname = 'Sales' AND wage > 300;
```

The following are two queries submitted by students. They are both different to the solution presented by the instructor, but both can be proved to be semantically equivalent to the instructor's solution query and are therefore considered correct (refer to Table 2).

Table 3: Two correct query solutions (SA1 and SA2) in CQ class and how they were evaluated

Submitted query	*sys*	*std1*	*std2*
`SELECT fname, lname FROM employee JOIN department ON dNo WHERE dname = 'Sales' AND wage > 300;`	L7	L6	L7
`SELECT fname, lname FROM employee E WHERE wage > 300 AND EXISTS (SELECT * FROM department D WHERE E.dNo = D.dNo AND dname = 'Sales');`	L7	L7	L4

The following query is an incorrect query answer to the above problem (QP1).

Table 4: An incorrect solution (SA3) in CQ class and how it was evaluated

Submitted query	*sys*	*std1*	*std2*
`SELECT fname, lname FROM employee E WHERE dname = 'Sales' AND wage > 300;`	L2	L6	L4

3.2 Problem Example 2

The next problem (QP2) involves a query that is not in CQ class.

> *List all locations where there is either an employee or a department. (QP2)*

The following is an instructor's solution query for this problem.

```
(Select elocation From employee) UN-
ION (Select dlocation From depart-
ment);
```

Table 5 shows an incorrect solution to this problem.

Table 5: An incorrect solution (SA4) in CQ class and how it was evaluated

Submitted query	*sys*	*std1*	*std2*
`Select loc FROM employee, department WHERE loc = elocation OR loc = dlocation;`	L2	L2	L3

3.3 Marking Query Correctness

When the system has evaluated a submitted query and peer reviews are complete for that query the system will recommend a mark to the instructor. The instructor can then assign an *accuracy mark* for the query. Table 6 shows, for each row, the correctness marks for a particular query submitted by a student, as given by the system itself (*sys*), and two peers reviewing the query answer (*std1* and *std2*). In addition, a suggested mark is shown calculated by *SQLify* on the basis of *sys*, *std1* and *std2*. Refer to [7] for details on how this is achieved. Finally, the *accuracy mark* assigned by the instructor is listed; this mark may or may not be the same as the suggested mark.

The internal values corresponding to levels given in Table 2 are not hard-coded into the system. The instructor using *SQLify* can set these values during use of the system. Hence, percentages given to query answers can be different in practice from the ones shown here.

Table 6: Correctness marks for submitted query answers

Student	Problem	Submitted query	System mark (*sys*)	Reviewer	Mark (*std1*)	Reviewer	Mark (*std2*)	Suggested mark	*Correctness mark* set by instructor
1	QP1	SA1	L7	3	L6	5	L7	L7	L7 (100%)
1	QP2	SA4	L2	4	L2	5	L3	L3	L3 (40%)
					…				
4	QP1	SA2	L7	1	L7	3	L4	L7	L7 (100%)
5	QP1	SA3	L2	1	L6	2	L4	L4	L4 (70%)

3.4 Checking Accuracy of Reviews

Table 7 lists one row per peer review that is performed in the context of an assignment. The first row, for instance, shows that student 1 was a reviewer for a query (SA2) submitted by student 4 in answer to query problem QP1. Student 1 gave this query answer a correctness mark of L7. The accuracy mark for the submitted query answer given by the instructor was also L7. Hence, the accuracy mark for this particular review is 100. For the next review performed by this student there is a difference between the correctness mark given by this student and the accuracy mark set by the instructor. This difference causes their mark for accuracy to be reduced.

Table 7: Accuracy marks for reviews

Reviewer	Reviewee	Problem	Submission	Reviewer's mark for submission	Accuracy nark set by instructor	Difference	Accuracy mark for this review
1	4	QP1	SA2	L7	L7	0%	100%
1	5	QP1	SA3	L6	L4	20%	80%
				…			

3.5 Calculating a Final Mark

The last table below summarizes the various marks that a particular student received for various query problems and for the reviews performed. A weighted final mark is given in the last row using the suggested weightings of 70% for correctness and 30% for accuracy of reviews.

Table 8: Final mark calculation

Student: 1		
Correctness marks	QP1	100%
(Weight 70%)	QP2	50%
	QP3	70%
Review accuracy	QP1	100%
(Weight 30%)	QP2	80%
	QP3	50%
Final Mark		74%

4. CONCLUSIONS

In this paper a small set of existing tools used for teaching and assessing SQL writing skills was reviewed. The tools were evaluated from both from Computing Education and Database Theory perspectives, noting possible areas of enhancement.

A new tool called *SQLify* was introduced which is used for practice and submission of database query assignments. Central to *SQLify* is the use of an intricate automatic grading system and of peer review. The main reason for including peer review is to offer the students a richer learning experience. Additionally, the peer reviews will assist in the assessment of assignments.

SQLify uses a relatively complex method to suggest marks for assignments, designed to:

- yield a much wider range of accuracy marks than simply *correct* or *incorrect*;
- employ *peer review* of assignment work by students encouraging evaluation and producing more sources of feedback to students;
- utilize *database theory* to enhance computer assisted grading;
- set high quality demands for student reviews, yielding higher learning outcomes; and
- reduce the number of necessary moderations by course instructors.

Each of these objectives must be made transparent to students. Students are informed of the possible learning benefits for students and the time-saving benefits for instructors. Students must be made aware of how the marking approach will be used to assess their work and their reviews and how they must use the system to succeed in assessments.

SQLify has been prototyped and implemented and is ready to be used in a live course by the end of 2006, with the exception of Relational Algebra support. Student use of the system will be monitored. The usefulness of the system as perceived by students and instructors will then be evaluated. Any change in student outcomes will be measured.

With this new tool it will also be possible to effectively distinguish specific problems within the areas of difficulty suggested in section 1.1, allowing feedback into the existing curriculum to improve teaching in these areas.

5. REFERENCES

[1] Bloom, B.S., *Taxonomy of Educational Objectives*. Edwards Bros., Ann Arbor, Michigan, 1956.

[2] Brook, C. and Oliver, R., Online learning communities: Investigating a design framework. *Australian Journal of Educational Technology*, 19, 2, 2003, 139 - 160.

[3] Chapman, O.L. *The White Paper: A Description of CPR*. 2006 [cited February 23, 2006]; Available from: http://cpr.molsci.ucla.edu/cpr/resources/documents/misc/CPR_White_Paper.pdf

[4] de Raadt, M., Toleman, M., and Watson, R. Electronic peer review: A large cohort teaching themselves? In *Proceedings of the 22nd Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE'05)*. (Brisbane, December 4-7, 2005). QUT, Brisbane, 2005, 159 - 168.

[5] de Raadt, M., Toleman, M., and Watson, R., An Effective System for Electronic Peer Review. *International Journal of Business and Management Education*, 13, 9, 2006, 48 - 62.

[6] Dekeyser, S. and de Raadt, M. *SQLify project website*. 2006 [cited May 15, 2006, 2006]; Available from: http://www.sci.usq.edu.au/projects/sqlify/.

[7] Dekeyser, S., de Raadt, M., and Lee, T.Y. *Computer Assisted Assessment of SQL Query Skills*. 2006 [cited 1st September, 2006]; Available from: http://www.sci.usq.edu.au/research/workingpapers/sc-mc-0610.ps.

[8] Dietrich, S.W., Eckert, E., and Piscator, K. WinRDBI: a Windows-based relational database educational tool. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (San Jose, California, United States, February 27 - March 1, 1997). ACM Press, 1997, 126 - 130.

[9] Kearns, R., Shead, S., and Fekete, A. A teaching system for SQL. In *Proceedings of the 2nd Australasian conference on Computer science education*. (Melbourne, Australia, 2 - 4 July 1997), 1997, 224 - 231.

[10] Kurhila, J., Miettinen, M., Nokelainen, P., Floreen, P., and Tirri, H. Peer-to-Peer Learning with Open-Ended Writable Web. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education ITiCSE '03*. (Thessaloniki, Greece, June 30 - July 2, 2003). ACM Press, 2003, 173 - 178.

[11] Mitrovic, A. Learning SQL with a computerized tutor. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education SIGCSE '98*. (Atlanta, United States, 25 - 28 Feb, 1998). ACM Press, 1998, 307 - 311.

[12] Prior, J. Online assessment of SQL query formulation skills. In *Proceedings of the fifth Australasian conference on Computing education*. (Adelaide, Australia, 4-7 February, 2003). Australian Computer Society, 2003, 247 - 256.

[13] Prior, J. and Lister, R. The Backwash Effect on SQL Skills Grading. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. (Leeds, UK, 28 - 30 June, 2004). ACM Press, 2004, 32 - 36.

[14] Sadiq, S., Orlowska, M., Sadiq, W., and Lin, J. SQLator: An Online SQL Learning Workbench. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education ITiCSE '04*. (Leeds, UK, 28 - 30 June, 2004). ACM Press, 2004, 223 - 227.

[15] Saunders, D., Peer tutoring in higher education. *Studies in Higher Education*, 17, 2, 1992, 211 - 218.

[16] Shneiderman, B., Creating creativity: user interfaces for supporting innovation. *ACM Transactions on Computer-Human Interaction*, 7, 1, 2000, 114-138.

Modelling Student Behavior in Algorithm Simulation Exercises with Code Mutation

Otto Seppälä
Helsinki University of Technology
PL5400, 02015 TKK, Finland
oseppala@cs.hut.fi

ABSTRACT

Visual algorithm simulation exercises test student knowledge of different algorithms by making them trace the steps of how a given algorithm would have manipulated a set of input data. When assessing such exercises the main difference between a human assessor and an automated assessment procedure is the human ability to adapt to the possible errors made by the student. A human assessor can continue past the point where the model solution and the student solution deviate and make a hypothesis on the source of the error based on the student's answer. Our goal is to bring some of that ability to automated assessment. We anticipate that providing better feedback on student errors might help reduce persistent misconceptions.

The method described tries to automatically recreate erroneous student behavior by introducing a set of code mutations on the original algorithm code. The available mutations correspond to different careless errors and misconceptions held by the student.

The results show that such automatically generated "misconceived" algorithms can explain much of the student behavior found in erroneous solutions to the exercise. Non-systematic mutations can also be used to simulate slips which greatly reduces the number of erroneous solutions without explanations.

1. INTRODUCTION

On the Data Structures and Algorithms courses in the Helsinki University of Technology we use the TRAKLA2 system[7] to assess how well students know how different algorithms taught on our course should operate. Rather than requiring the students to implement these algorithms, the system tests their knowledge using *visual algorithm simulation exercises*. These exercises are then automatically graded and form a part of the course grade. While automatic assessment saves us hours and hours of assessment time it also gives the students the possibility of getting immediate feedback on their solutions during day and night.

This far the feedback has typically consisted only of the number of correct steps and a model solution. As the student solution is also still available, an interested student has been given a possibility to review the answer against the solution and figure out what went wrong. For some time now we have been researching on how to improve the quality of this feedback and essentially it all boils down to being able to interpret the error made by the student.

Our previous paper[8] on the subject studied the possibility of simulating the errors by manually implementing algorithm variants that correspond to different misconceptions. The approach described in this paper extends on this work with a way to automatically generate some of the algorithm variants as well as a method to handle careless errors.

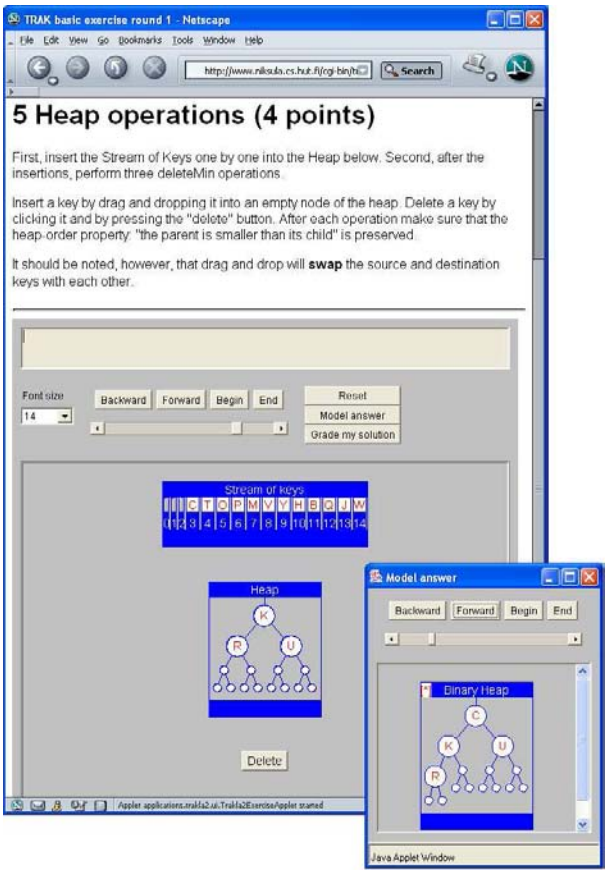

Figure 1: TRAKLA2 applet page and the model solution window. In this exercise the heap operations can be simulated by moving the keys in the data structures using a mouse.

2. VISUAL ALGORITHM SIMULATION EXERCISE ASSESSMENT

A visual algorithm simulation exercise typically includes a number of data structures in randomly generated initial states and an exercise description which explicitly tells which algorithm to follow. Depending on the exercise being solved, the student can use use the mouse to perform different drag and drop operations which again swap keys, change edges in dynamic structures etc. Each operation that changes a data structure is recorded to the *student solution sequence*. This sequence is later on graded by checking it against a *model solution sequence* generated by an existing implementation of the algorithm.

The normal method of assessing these exercises is to first select a state from the model solution sequence and then browse through

the student sequence until a matching state is found. The comparison process then advances to the next state in the model sequence and searches for a matching state in the student sequence from the states following the last match found. The process is repeated until we run out of states in either sequence. The points from the exercise are proportional to the amount of model sequence states matched by states in the student sequence.

Typically the model sequence is shorter than the student sequence. In some cases the student has freedom to choose in which order to perform some minor operations, the order of which is not explicitly stated in the algorithm. A swap operation for example allows two different orderings. To avoid useless iteration we can allow all such orderings by only requiring the states where all minor operations have already been performed.

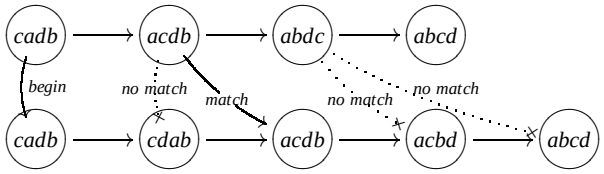

Figure 2: Comparison between two sequences. (Selection-sort of an array) The top sequence is the model, the bottom the student sequence. Two matches were found.

One limitation of the current assessment procedure for the visual algorithm simulation exercises is that while it can count the number of correct steps is a sequence, it offers no information on what was the cause behind the incorrect step in the algorithm simulation. It also cannot find out if any steps following such erroneous state could have been correct relative to the incorrect state created by an earlier mistake made by the student.

When similar algorithm tracing exercises are solved in a paper exam, a human examiner can often tell if the errors are just random slips or more systematic errors possibly resulting from misconceptions. Telling the two apart is important for giving students proper feedback. A human examiner can also continue the assessment past a small careless error. One aim of our research project is to try and bring some of this ability also to automated assessment.

3. INTERPRETATION OF ERRORS

After the student has submitted her solution she can view the model sequence for that exercise instance and compare where and how the answer sequenced deviate. As explained, the problem with interpreting the model solution sequence is, that while locating the differing state in the sequences is easy and straightforward, finding out the reason behind this difference is not. For that, a student (or the examiner) should actually repeat solving the exercise. This is because the errors that were made, were made in a specific *context* created by the previous steps of the algorithm – to interpret them, that same context is needed. Analogously, if you begin reading a novel from the middle, understanding the next action in the plot is often somewhat impossible.

The same is essentially true when trying to reinterpret a recorded student solution or even an answer written on an exam paper. As the recorded data structures do not include many of the auxiliary variables used nor the position in the algorithm code, the assessor must also start the tracing of the algorithm from the beginning to understand the error made by the student.

3.1 Automatic Context Creation in the Presence of Misconceptions

Artificially recreating the interpretation context seems a fairly straightforward task. An implementation of the algorithm can be run and its data structures compared until we find the last state that matches the recorded sequence. At that time the implementation holds the "full" algorithm state with also the variables that were not originally recorded in the student sequence. Such variables include loop variables etc. which are not shown or input to the system by the student. The reconstructed context can then be used to better evaluate the error the student made.

As the context is directly dependent on the algorithm being traced, it is also affected by any misconceptions the student holds about that algorithm. This again requires that contexts are created for misconceived algorithms as well.

In [8] we tested if student misconceptions about algorithms could be modeled by manually implementing *variants* of the algorithm being studied. The misconceptions could then be identified by using the same assessment method that is used to assess the student solutions. Essentially, the user solutions were tested against each of the variants to see if any of the sequences created by the variants better matched what happened in the student sequence. The results from the study showed that the approach is quite usable for pointing out popular misconceptions. An exhaustive search for less popular variants and implementing them however seems not feasible or is at least very laborious to do manually.

While the approach described is able to find exact matches to any of these variants, it cannot recognize or handle *careless errors*, which are often only small alterations in the form of skips, off-by-one errors, misreading alphabetic order and such. It would however be possible to model skips by creating a separate variant for each and every skip or combination of skips. It is quite clear that manual implementation can not be done given the number of possible combinations.

Even if we do not consider the careless errors, manually implementing each prospective candidate also takes a lot of work. The tools used for tackling both of these problems to a certain degree and automatically creating algorithm variants are introduced in the next section. In section 5 we define two assumptions which define the area of applicability. Section 6 describes the actual implementation used. Results from an experiment on recorded answer sequences are given in section 7. Section 8 discusses future work.

4. RELATED WORK

An approach to recognizing misconceptions by making alterations on a model of a skill was used by the BUGGY[2] and DEBUGGY[4] systems. The systems tried to infer misconceptions held by pupils learning the basics of in-place subtraction. The system had a model of subtraction that was divided into subskills that could be replaced by their incorrect counterparts. If the subtraction problem was carefully selected, a matching result generated by the altered model could point out students who for example had a specific misconception of how to borrow.

Another influencial system to be mentioned is the LISP Tutor by Anderson et al.[1]. LISP Tutor had a model that would perform the task the student was expected to perform. While the student solves a problem, correct and incorrect steps are immediately recognized. In case of errors the student is given instruction that tries to steer the student back to a correct path. Their original solution to interpreting what the steps made by the student actually meant was to provide the student with a disambiguation

menu, which provided the model with information if could not infer from the students actions.

5. CODE MUTATION

The automatic variant creation described in this paper is done by introducing changes in the code of the original implementation[1]. Such controlled changes are often used in applications such as *mutation testing*, *fault injection* and *genetic programming*. The methods we will use for creating the algorithm variants have been originally introduced and extensively studied for use in mutation testing. As a result we will also use some of the vocabulary in that area.

5.1 Background

Mutation testing is an idea proposed by DeMillo et al,[5]. It aims at evaluating and improving test data by introducing changes, *mutations*, to the program code which the test cases should then be able to find. A test case capable of killing an introduced bug is potentially able to find other bugs in the bug's neighborhood.

5.2 Mutant Creation

The changes to the program code are made with *mutation operators*. A mutation operator is a change which typically replaces a small portion of the program code with different code. A classic example would be to interchange any of the arithmetic operators $+ - / *$ with another arithmetic operator at one point in the code. A program changed with at least one mutation is called a *mutant*.

An instrumented code that contains all the possible mutations controllable at runtime by the testing system is called a *metamutant*[9]. The main advantage of this approach is that the code is only compiled once. The downside is that the the code executes slower. In the prototype described in this paper the metamutant approach is used. We will use the name *mutation point* when referring to portion of code in a metamutant which is changeable with a mutation operator.

6. ASSUMPTIONS

The misconception modeling approach makes two assumptions of the students and their knowledge which are essential for this approach to work. The assumptions both define the area of application and are also used when pruning the search tree in the mutation search for the best candidates.

6.1 Systematicity Assumption

It is quite safe to assume that university students know at least one thing about the algorithms taught on the course – that algorithms are executed in some systematic way. Therefore even in the presence of a misconception it should hold that the whatever algorithm the student is trying to follow, it would still be systematic. We just do not know which algorithm it is. Although there exist students that might not share this understanding, we do not have to consider them as their problems are too profound to be tackled with algorithm simulation exercises.

Exceptions to this systematicity rule are the unintentional careless errors made. After the careless error the sequence should continue with normal steps created by the algorithm, be it the correct or a misconcepted one.

It is also important to point out that as the TRAKLA2 exercises often require the student to repeat the algorithm on a set of data instead of a single key etc., the systematicity of possible misconceptions shows up quite well even for algorithms that would normally have a relatively small number of steps.

[1]and possibly also the code of any manually implemented variants

6.2 Mutation Distance Assumption

The second assumption is that as many of the misconcepted algorithms are derived from the original algorithm, the implementation of the misconcepted algorithm is not far from the implementation of the correct algorithm.

In a sense this is related to the *competent programmer hypothesis*[3] which is one of the cornerstones of mutation testing. The hypothesis is that competent programmers should be able to create programs that only differ from a perfect program by a given distance. The hypothesis is essential for the mutated program to efficiently model the real-world faults the test cases should be able to catch.

We know however from data collected from students that the mutation distance assumption does not always hold. Manual search through the answer sequences has shown that misconcepted algorithm variants exist that have notably more complex implementations than the original algorithm. The original algorithm might for example require no external storage whereas the student algorithm might require a dynamic memory structure such as a stack to be implemented.

This is not surprising as the synoptical view onto the data structure easily misleads students uncustomed to working with data structures by hiding the inherent complexity behind a seemingly simple approach to a problem. A good example is performing a sort, which can be performed by anyone, with or without any education in sorting algorithms.

This finding only implies that the mutation methods must be backed up with some manual implementation of the more distant algorithm variants. Mutation can then be used to find the subvariants in their vicinity.

6.3 Implications of the Assumptions

In this research we concentrate on the misconcepted algorithms that fill both of these assumptions. It is clear that if the systematicity assumption does not hold, the sequences generated are uninteresting as randomly trying out the algorithm typically is not a sign of a misconception about something learned but more of a wild guess. There is no point in generating guiding feedback from guesses.

For the second assumption to hold, the first one must already be true. As previously pointed out, the second assumption can be violated but the misconcepted algorithms could still be systematic.

We have demonstrated in [8] that it is possible to sieve out likely candidates for misconcepted algorithms and then implement these by hand. In this paper we however are interested in algorithms that can be derived from the original with more minute changes.

The next section explains the mutant creation and mutation search procedure in more detail.

7. IMPLEMENTATION

For the prototype we have chosen to use the metamutant approach. The code for the metamutant cannot currently be created automatically. The algorithms are therefore prepared by hand, inserting the *mutation points* where appropriate. Knowledge on the algorithm operation is of assistance when choosing the mutation points as all mutations do not lead to sensible code, but only create useless execution.

The series of mutation choices made at each mutation point when executing the metamutant is called a *mutation sequence*. This is not to be mistaken with the student sequence and the model sequence which store the states of the data structures. The process of finding the mutation sequence which best explains the recorded student sequence is called a *mutation search*.

7.1 Mutation Operators

A *mutation operator* is a description of a syntactically correct change to an existing program that will change the semantics of that program, resulting in a new program, called a *mutant*

Typically mutation operators are designed to make single-point changes to the source code of the program. The changes can be made prior to compilation, or at runtime if the code has been instrumented to host a number of mutations that can be switched on and off on demand.

Our approach is different from the normal use of mutation operators, as we want to change the state of the mutation run-time. This is required to model the careless errors that break the systematicity of the algorithm.

For this prototype implementation we have chosen to use only a small number of mutation operators, which are decribed below.

- *Zero Operator* is a restricted form of the more general integer offset operator used by many mutation testing systems. It normally evaluates to zero, but can also evaluate to one, making it useable for modeling off-by-one errors, skips etc. depending on the place where it is applied. Respectively there also exists a ”one”-operator.
- *Comparison Operators* change the result of the corresponding boolean comparison to its negation. Although we normally have 6 different comparison operators to choose from, the mutation operator works perfectly well working only with the original form and its negation. This limits unwanted forking of the mutation search tree.

  Later on, when we want to investigate the final candidates, we can infer from the variable values, which of the comparison operators would have returned true or false when the comparison was made.
- *Arithmetic Operators* change additions to substractions and vice versa. Divisions can be rounding instead of truncating. Using a rounding division is a fairly common novice mistake.
- *Skip Operator* essentially is a boolean value used to control a conditional clause which can be used to skip a portion of the code.

The points in the code where steps are added to the model solution sequence are marked with code similar to the mutation points. These code locations, *Animation points*, are important for pruning (explained in 7.4), as they are the only possible locations where new steps are added in the solution sequence and are thus the only places where the score of the comparison between the student sequence and the generated one could increase.

7.2 Mutation Recorder

The mutation recorder is used to record the mutations made during the execution sequence. Each time a mutation point is passed, it creates a new link, *mutation step*, in the mutation sequence. Such links are also made when the mutation point operates like the original non-mutated code. The specific mutation is chosen by the mutation recorder based on the previous mutations taken that followed this same path.

The information on the child nodes expanded is stored in the mutation step objects as well as which child of the father node this node is. All the mutation sequences together form a father-linked tree where each mutation operation links to the previous one. This approach not only reduces memory consumption, but also allows a mutated execution sequence to be referenced only using the link created by the last mutation operation.

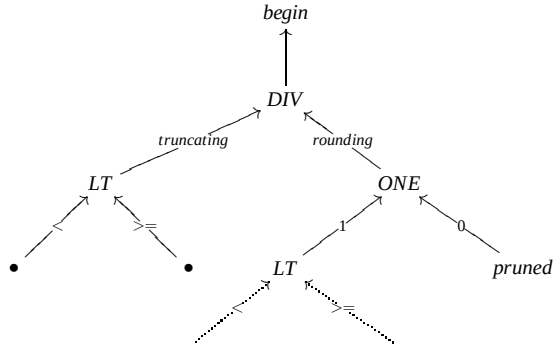

Figure 3: A tree containing mutation sequences

The recorder is also used to replay the changes when a variation of a previous sequence is wanted. This functionality is required when we want to derive a new sequence from a previous one. In this case a part of a mutation sequence is rewound into a stack inside the mutation recorder. When replaying, the mutation points act the same way as when the original sequence was recorded. When the steps in the stack are finished, the recorder goes back to its normal operation.

7.3 Search Algorithm

The search algorithm attempts to generate an algorithm variant through mutation, which best explains the student’s answer sequence. This is a simple depth first search with backtracking. Pruning the search tree nodes is used to make the search both feasible and also faster. The search algorithm finds first an initial solution, be it full or partial[2]. Typically the first try uses the original, unmodified algorithm. It then backtracks until a link is found with children that have not yet been expanded. Path from the start to this node is rewound to the mutation recorder for playback as explained in the previous paragraph.

The solving process is repeated with the steps stored in the recorder. When they end, the next node always is a node with unexpanded children. The recorder then selects a new child and the solving process continues expanding new nodes until the procedure is stopped the next time. The stopping conditions are given in the next section. The search algorithm will go through all possible mutation sequences up to the normal ending point of the algorithm or a point where the search is pruned or ended early by an exception.

7.4 Pruning the Mutation Search Tree

If the mutation search is performed without limiting the search in any way, the search would never end. Not only is the amount of branching between the beginning and the end very high even for a normal case, but there also exist mutated execution sequences that result in non-terminating execution. The mutations can also

[2]Partial solutions do not explain all student sequence steps, because they are pruned

lead to exceptions or early termination which are considered here as forms of "self-pruning". These are however considered to be a positive side-effect as they are quite effective in reducing the search space.

A partial solution to the problem of high branching and endless execution exists in the form of two pruning operators that have proven to be quite effective:

- *Limiting inconsistency in mutations made in a single execution*

  This pruning rule is a straight consequence of the systematicity assumption. If a mutation operator constantly changes its function, the algorithm is not anymore systematic. A limited number of deviations are allowed which model the careless errors done by a student.

- *Limiting the number of changes to the data structures which do not lead to a higher score*

  As with the traditional checking algorithm used by TRAKLA2, we have to take in account that the model algorithm only contains the major states between which the minor states can be scrambled. Scrambled or not, a limited number of changes to the data structures checked should lead to the next major state. This again should raise the score gained from the exercise.

  This pruning rule stops the search in a branch that has too many intermediate states following a "score state" that produce no change to the score. It is important to note that while the number of such intermediate states is specific to each algorithm, the value typically is quite low and controllable by the exercise designer.

7.4.1 Exceptions

It is not uncommon for the mutated algorithm to crash with an exception. Mutations violate the assertions the original algorithm follows. As the mutated algorithm implementation is seen as a model of the student's simulation, the exception thrown in the program code is essentially also an operation that could not be simulated. Sequences leading to exceptions can therefore be pruned.

7.4.2 Normal Termination

If the mutated execution is not pruned or does not end with an exception the result can either be a successful interpretation of the simulation sequence or an early termination, in which case we can prune the ended sequence if there was earlier an other sequence with a higher score. All terminating sequences are therefore evaluated and the best candidates chosen for closer inspection.

7.4.3 Other Cases

There exist cases that are not pruned by either of the proposed pruning operators, do not normally terminate, and do not throw exceptions. Such cases fall into three categories of non-terminating executions.

1. non-terminating execution path with animation points
2. non-terminating execution path with mutation points
3. non-terminating execution path with no mutation points

The first two could be pruned using similar pruning filters which are checked when a mutation point is reached. Even then, deciding the level of when to prune is not easy. For example, if a sorting algorithm is executed on an already ordered data structure, there would still have to be at least $O(n)$ comparisons made just to verify that the array is sorted. It is likely that there would be at least this amount of animation points passed where there are no changes to the data structure. Correspondingly there should be an even higher number of mutation points passed on the way, where pruning should not be done.

The last category, non-terminating execution path with no mutation points, cannot be pruned using any pruning techniques introduced this far, as pruning is only made in the annotated parts of the code.

The prototype therefore currently requires the programmer to recognize such points and write assertions that throw exceptions or force the code to eventually end in such situations.

8. RESULTS

To evaluate the approach used, a set of real recorded solution sequences from our data structures and algorithms course was used. The effectiveness of the mutation-based method was compared with the method used in [8]. The exercise used in the study was binary search as hand-implemented variants of misconceptions on that exercise already existed. These variants modeled misconceptions about truncating division and movement of the left and right pointers in the algorithm.

The hand-implemented variants were able to explain 40% of the sequences that were nor fully correct or completely empty (nothing done). For the mutation method the amount of solutions with at least one explanation was 65%. The improvement was mostly from the inconsistent mutations which were not possible to model in the manually implemented variants. The consistent mutations are in line with the hand-implemented variants although for some solutions the mutation method was able to find a simpler explanation using a single inconsistent mutation in place of two consistent ones.

It is important to mention though, that the binary search exercise is exceptional in the sense that all the hand-implemented algorithms were found using consistent mutations. For most algorithms it is likely that we still need to implement many of the variants by hand. The mutation approach can then be used to find minor deviations from these algorithms and to recognize slips.

8.1 Effect of Pruning

Both the two pruning strategies are of importance if we want to provide the student with immediate feedback. If no pruning is used the combinatorial explosion ensures that finding a solution is not feasible for a reasonable-sized exercise instance. Allowing one slip and one additional step made by the solution algorithm cuts the mutation search time down to 3 seconds per solution. Allowing more errors in the sequence might multiply the time by ten for each new error allowed. The pruning values are dependent on the exercise instance. The effect on the values on the solutions found is a matter of another study.

8.2 Known Limitations

One known limitation of the approach is that in many cases some seemingly simple systematic changes done by the students violate the mutation distance assumption. A good example of this is was found in the binary tree in-order traversal exercise. The normal in-order traversal recursively traverses the left subtree, then visits the node itself, and then traverses the right subtree. In some student sequences the node is visited after the right subtree if the left tree is missing. It is possible that not having the left subtree

contradicts with a simplified model for traversal: left-node-right. The right subtree is then used in place of the left.

Such a simple-looking change adds to the complexity of the traversal algorithm in a way that cannot easily be handled with mutation. One could argue that a mutation operator capable on reordering code lines would be able to generate the desired mutant, but even then the mutation had to be conditional, depending on the value of the reference to the left subtree.

It is therefore likely that the main variations of the algorithms must still be prepared manually, but combining the manual approach with the mutations is a promising tool for examination of the solutions.

9. CONCLUSIONS

When students solve algorithm simulation exercises, the solutions form a sequence of states. Previously we have shown that a considerable number of these sequences can be explained by corresponding algorithm variants that model some typical misconceptions. We have now described an automatic way of generating variants from existing algorithm implementations through code mutation. This is an advantage over the previous results, as allows for more easily creating a variety of different versions of the original algorithm. It also allows modeling of random carelessness errors with non-systematic mutations.

Initial results suggest that introducing the carelessness error to the modeling sometimes allows for simpler explanations of students' simulation sequences.

10. FUTURE WORK

The next logical steps are to enhance the analysis of the results and correspondingly also the feedback to the students. Secondly the quality of the system and the feedback should be evaluated with students. It should also be possible to create the metamutant automatically.

10.1 Enhanced Feedback

One of the most interesting parts of the project comes when the error is presented back to the student. The aim is to use a visualization of the pseudo-code of the algorithm to point out the place where the student execution diverged from the model solution. This information visualized is actually the types and places of the mutations made to the code.

Another possibility is to write textual feedback that matches pattern of mutations found by the prototype. The problem with this is the possible conditionality of the mutation – that the mutation happens only when a specific condition is true. Therefore in most cases it seems that showing the position in the pseudo-code along with the type of change made by the mutation operator seems as the best alternative for now.

10.2 Evaluation

As the results of the trial runs with real course data have been promising, the next logical step is to evaluate the quality of the results online. Although the design of the evaluation is open, one possible way could be to first use multiple choice questions to probe the knowledge and misconceptions held by the student. These questions would be presented immediately after the user has submitted a solution, but before showing the exercise results. After the initial questions have been answered, we can show the results and the automatically generated feedback, the quality of which the user can then evaluate. Also, as the student would not be aware of any slips before seeing the results, an additional question about them would be presented as well.

10.3 Heuristics for Breaking Ties

Heuristics for selecting the best candidate are required when there are multiple good mutation candidates that equally well match the sequence. Such decisions are often case-specific, e.g. whether two consistent mutations are better than one inconsistent, depends a lot on the exercise instance and the specific mutation operators.

10.4 Tools

Automatic creation of the metamutant requires a simple Java parser to be built. As there is no actual need to understand the semantics of the code, this should be possible to do with only a little more than simple pattern matching and replacement. The additional requirements are assuring that the code is syntactically correct and checking that the mutation point must also operate in the exact same way as the original code did.

The infinite execution problem currently requires attention from the exercise designer. The placement of the assertions that break the infinite loops should also be possible to do automatically. This requires recognizing iterations and recursions in the code that fall in the three categories decribed in 7.4.3, and placing assertions in the loops that at least calculate and limit the number of iterations made.

11. ACKNOWLEDGEMENTS

This work was supported by the Academy of Finland under grant number 210947.

12. REFERENCES

[1] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier. Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995, Lawrence Erlbaum Associates, Inc.

[2] J. S. Brown and R. B. Burton. Diagnostic models for procedural bugs in mathematical skills. *Cognitive Science*, 2:155–192, 1978.

[3] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM Press.

[4] R. B. Burton. Debuggy: Diagnosis of errors in basic mathematical skills. In *Intelligent Tutoring Systems*. Academic Press, 1981.

[5] R. A. Demillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–41, 1978.

[6] A. Korhonen. *Visual Algorithm Simulation*. Doctoral dissertation (tech rep. no. tko-a40/03), Helsinki University of Technology, 2003.

[7] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267 – 288, 2004.

[8] O. Seppälä, L. Malmi, and A. Korhonen. Observations on Student Misconceptions - A Case Study of the Build-Heap algorithm. *Computer Science Education*. 16(3):241 – 255, September 2006, Routledge.

[9] R. H. Untch. Mutation-based software testing using program schemata. In *ACM-SE 30: Proceedings of the 30th annual Southeast regional conference*, pages 285–291, New York, NY, USA, 1992. ACM Press.

Discussion Papers

Learning Programming by Programming: a Case Study

Marko Hassinen*
Department of Computer Science,
University of Kuopio
P.O. Box 1627
70211 Kuopio, Finland
marko.hassinen@uku.fi

Hannu Mäyrä
Department of Computer Science,
University of Kuopio
P.O. Box 1627
70211 Kuopio, Finland
hannu.mayra@uku.fi

ABSTRACT
Programming is a challenging field of computer science for both to teach and learn. Although studied extensively, a definite method for teaching programming is yet to be found. In quest of finding success factors in both elementary and more advanced programming courses, this paper discusses some findings made studying exam success and home assignment activity in programming courses. Our claim is that there is no shortcut in learning to program, but extensive practise and sufficient time to become familiar with programming concepts is needed.

Keywords
Learning programming

1. INTRODUCTION
Programming is probably the most challenging field of computer science to teach [2]. Studying proramming involves not only learning new things but more importantly learning how to apply this new knowledge to solving problems. The ability to see how things are connected to each other and derive new constructs from existing ones is utterly important for a programmer. Terms like knowledge, understanding and expertise can be used to differentiate the abilites needed to program a computer.

While knowledge can be gathered by reading books or attending lectures, understanding requires deeper processing of the knowledge and, in most cases, hands-on practise [1]. We claim that only through adequate practise and training can expertise be obtained in the field of programming. To support this claim we evaluate results of two programming courses and study the importance of practise in students success on these courses. In the light of our findings we discuss two methods for measuring learning on programming courses, namely the traditional paper exam and a novel approach where exam assignments are pure programming assignments carried out with a computer. Both of these methods are contrasted with a study of how student activity in completion of practical assignments during courses affect their success in exams.

2. METHOD
The material was collected at University of Kuopio from elementary programming course for first year computer science students in 2001 and 2004 as well as advanced network programming courses for third and fourth year students in 2002 and 2005. On the elementary programming course the programming language was Java, and on the network programming course there were several programming languages, such as Java, PHP, and Perl. The material contains a total of 226 records, 153 from the elementary course and 73 from the advanced course.

The students were given assignments which they were expected to solve by a certain date. Excercise sessions were held on these deadline days, where each student marked which assignments he or she had solved. Students who solved more than 50% of the given assignments were given an extra point to the final grade and students who finished more than 75% received two extra points to the final grade. The grade scale was from three to twelve. A threshold of 30% was kept as a requirement for taking the final exam. A student not meeting the threshold had to pass an additional exam to compensate for the missing exercises. As solved assignments gave students extra credit, marking solved assignments was controlled by random checks.

Individual learning results were evaluated with exams. The students could pass the course either by taking two small exams, one in the middle of the course, another one in the end, or by taking the final exam. The smaller exams had two formats, either a traditional paper exam or a computer exam, in which a student was given a programming task to solve and a computer to do it with. The expected result was a computer program solving the given task. The students were given the opportunity to take either the paper exam or the computer exam or both, in which case the better result was considered in the overall evaluation.

Feedback was collected at each course. On the elementary courses, feedback was collected twice, first time in the middle of the course and a second time in the end of the course. On the more advanced network programming course feedback was collected at the end of the course. The feedback form consisted of free form questions with an open question for any comments they wanted to give to either the lecturer or the teachers responsible for the excercise sessions.

3. RESULTS
Figures 1 and 2 depict how completed assignments reflect to the final grade. Figure 1 contains all 226 observations and their average plotted on the gray curve. Figure 2 shows the grade distribution averages of the two courses on separate curves. The material contains all students who got a grade from the course. However, students who failed to pass the exam are not included. Also, there is no information about failed exams of students who failed the first exam, but passed a subsequent retake. Unfortunately

*Corresponding author

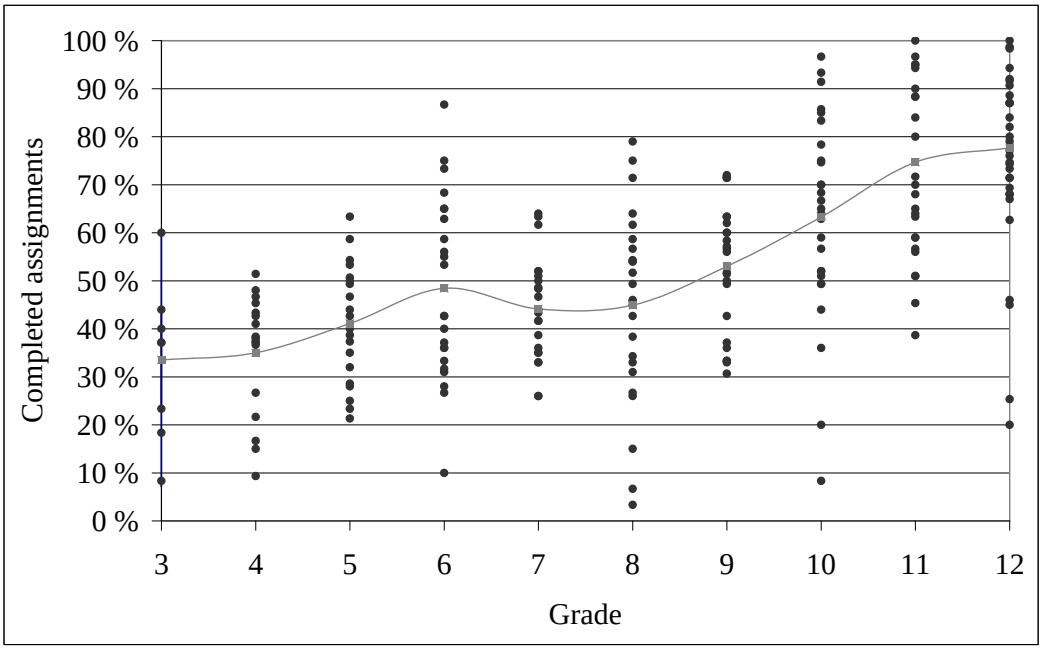

Figure 1: Distribution of completed assignments in Programming I and Network programming courses

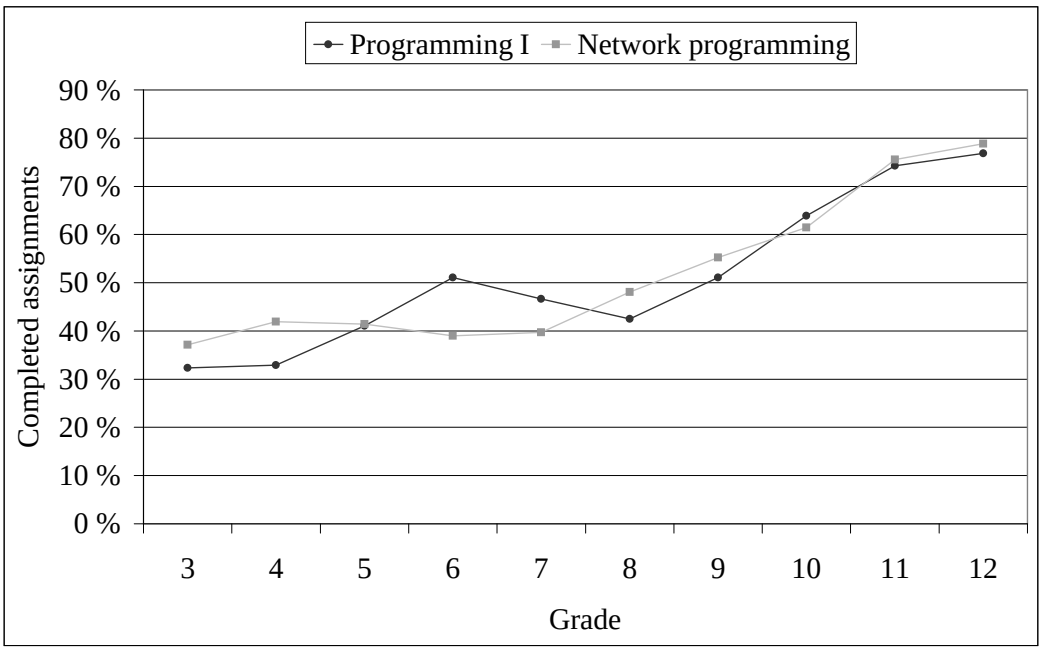

Figure 2: Average on completed assignments per grade

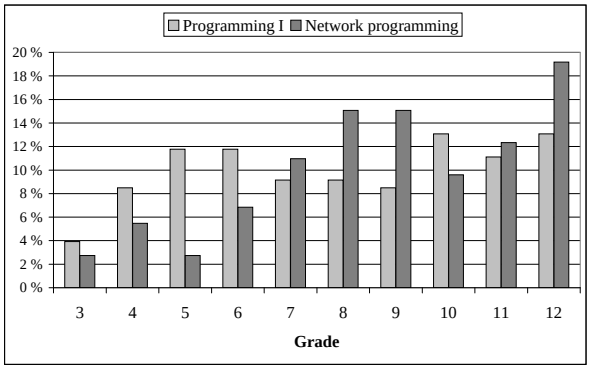

Figure 3: Distrubution of grades

the information on failed exams was no longer available.

From analyzing the material it became obvious that there is a strong positive correlation between the number of completed assignments and exam success. Students who had finished more than 75% of the given task scored excellent marks almost without exception. On the other hand, students who completed just the amount of assignments required to take part in the exam mostly scored low grades. However, the low assignment score had relatively more good exam results than there were poor results in the high assignment score group. This was an expected finding, since there obviously is a very wide range of different levels of knowledge in programming and general computer science among the first year computer science students. Students who already had programming knowledge and experience could pass the course with considerably smaller effort than those who started from the basics.

A very interesting observation can be made from Figures 1 and 2. It seems that both graphs present an s-shape in which there is a clear drop on exercise activity with grades six to eight. It seems that students who scored low grades had been more active in completing their assignments than those who scored mid range grades. This phenomena was present in both the elementary programming course as well as in the advanced programming course. However, the drop was considerably more moderate with the advanced course. A definite explanation could not be found.

Grade distribution is depicted in Figure 3. Extra credits given for those who completed more than 50% of the assignments partly explain the high percentages with the excellent grades.

The first and most evident observation from the results of the feedback was that as students are individuals, there seems to be no way to tailor a programming course that would suit each student optimally. The question about the pace of the course received a nearly equal amount of answers stating that the pace was too fast as those stating a too slow pace. Fortunately, the mid range that were happy with the pace was the majority.

Feedback from students supports also our observation; a large group of students answered that solving home assignments supported their learning experience the most.

4. OPEN QUESTIONS

Our study opened following interesting questions:

- How could the drop in exercise activity in the six to eight grade range be explained?
- What should be the ratio of excercises and lectures on the first programming course? Should this ratio change with more advanced courses?
- How do you motivate students in doing the assigned tasks and to invest the time it takes to complete these assignments?
- What is the impact of providing sample solutions to assigned tasks? Should one do that at all?
- It takes very different amounts of time for the students to finish a given assignment. While some students can solve easy assignments in matter of minutes, it may take hours for others to do the same. Is there any way to bridge this cap or is that even necessary?

5. CONCLUSION

Our findings clearly support the learning-by-doing aspect of studying programming. It can be seen that scoring excellent marks without extensively practising is not possible. In our study exceptions in this were students who had moderate to extensive programming experience already before the course.

6. REFERENCES

[1] R. Bruhn and P. Burton. An approach to teaching java using computers. *The SIGCSE Bulletin*, 35(4):94–99, 2003.

[2] I. Milne and G. Rowe. Difficulties in learning and teaching programming - views of students and tutors. *Education and Information technologies*, 7(1):55–66, 2002.

Is Bloom's Taxonomy Appropriate for Computer Science?

Colin G. Johnson
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
England
C.G.Johnson@kent.ac.uk

Ursula Fuller
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
England
U.D.Fuller@kent.ac.uk

ABSTRACT

Bloom's taxonomy attempts to provide a set of levels of cognitive engagement with material being learned. It is usually presented as a generic framework. In this paper we outline some studies which examine whether the taxonomy is appropriate for computing, and how its application in computing might differ from its application elsewhere. We place this in the context of ongoing debates concerning graduateness and attempts to 'benchmark' the content of a computing degree.

1. INTRODUCTION

Bloom's taxonomy was devised in the 1950s as a generic instrument for dividing the cognitive aspects of learning into hierarchical levels. It is now widely used in course design in higher education, as a way of ensuring that teaching and assessment strike the right balance between rote learning of content and high level skills such as synthesis and evaluation. The application of these cognitive levels now goes far beyond the design of individual modules [1]. Its influence can also be seen in attempts to define 'graduateness': what a student should be able to do at the end of a Bachelor's or Master's degree. Such specifications underpin the European Higher Education Area (EHEA)'s drive to ensure the international recognition of qualifications and the mobility of labour. The Bologna Declaration [8] has resulted in major higher education curriculum reform across most European countries and generic statements of competence at the end of the first, second and third cycles. There is an ongoing process, known as the Tuning Project [1], which is generating EHEA-wide, subject-specific statements of competencies akin to the UK's subject benchmarks [2].

A departmental attempt to improve assessment led the authors of this paper to apply Bloom's taxonomy to a number of first year modules and to wonder whether the ordering in its hierarchy is appropriate for computer science. This paper outlines our study of practice in a single university, and throws the question of the aptness of Bloom to computer science open to wider debate.

2. LEARNING TAXONOMIES

The learning taxonomy devised by Bloom et al [5] divides the cognitive aspects of learning into six hierarchical levels:

- Knowledge (recall of facts, et cetera)
- Comprehension
- Application
- Analysis
- Synthesis
- Evaluation

Bloom et al were somewhat equivocal about whether evaluation should be above or on the same level as synthesis and they were also not dogmatic about whether evidence of performance at a higher level necessarily demonstrated performance at all the lower levels.

There appear to be many interpretations of this taxonomy. Some teachers see the hierarchy as applying to individual topics. Every topic is capable of being approached at each of the levels, and the more successful the student is the higher the level she or he will reach. An alternative idea is that the hierarchy represents progress through the subject as a whole, for example in a degree programme. Under this interpretation, the lower levels correspond to early years of study, with the final aim of the programme being that all students will be enabled to achieve at the highest level.

Recent re-evaluation of Bloom's taxonomy by Anderson, Krathwohl et al [3] has suggested that the top two or three levels of the hierarchy may be flat (Figure 1). They have also proposed that the taxonomy should be two dimensional, with the (slightly reconfigured) original categories of Remember, Understand, Apply, Analyze, Evaluate and Create forming the cognitive process dimension and Factual, Conceptual, Procedural and Meta-Cognitive forming a knowledge dimension.

Whilst Bloom's taxonomy of the cognitive domain has the widest currency, it is not the only such taxonomy. For example, Bloom and his colleagues produced a much less well known taxonomy of the affective domain, while Biggs' SOLO taxonomy [4] charts increasing structural complexity in student learning outcomes. This identifies that learning first changes quantitatively, as the amount of detail in the students response increases, and then qualitatively, as the detail becomes integrated into a structural pattern.

The computer science education literature contains a small number of examples of the use of a taxonomy as an analytic tool. Bloom's taxonomy has been applied in course design; for example Scott [9] and Lister & Leaney [6] have used it for structuring assessments. Taxonomies have also

[1] In this paper we use the term module to denote a unit of learning that is assessed as a whole and might, typically, constitute a quarter, eighth or tenth of a year's study for a full-time student

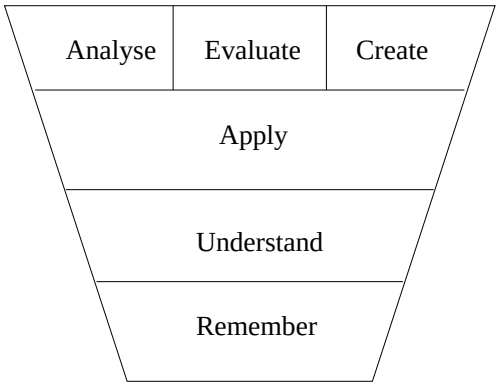

Figure 1: Bloom's Taxonomy 'flattened' [3].

been applied retrospectively, for example Lister et al [7] used the SOLO taxonomy to classify free-form responses to a problem-solving task

3. A STUDY OF ASSESSMENTS

A study was carried out which looked at all 54 assessments that were given to the first year students studying Computer Science in our university during one year. These were examined by a panel of five academics from the department (some of whom had been involved in these parts of the course, some not), who were asked to decide which of the levels in the Bloom taxonomy were being assessed by that particular assessment. The results are presented in Table 1.

4. INTERVIEWS WITH COURSE LECTURERS

A structured interview was held with the lecturer who was responsible for organising (and teaching a large component of) each of the first-year modules. As part of this interview, the lecturer was asked about the use of the various Bloom levels, whether they were relevant both to the material taught in the module, and, more specifically, whether they were evaluated as part of the module.

This part of the interview was introduced with a preamble about how learning can involve different levels of understanding according to the material being learned, and that assessment can emphasize these different levels.

Table 2 contains the questions, a sample of answers, and a summary of how many modules assessed material at this level.

5. COMMENTS AND OBSERVATIONS

A number of observations can be made from our study of assessment in first year computer science modules. The first is that there is considerable disagreement between the academics responsible for the design and delivery of these modules (conveners) and the group who analysed all the assessment tasks (assessors) about the level at which assessment was being carried out. The assessors felt that the vast bulk of assessment was at the *application* level, while conveners considered that they were also assessing *analysis*. One reason for this could be the difficulty of determining the taxonomic level of the assessment without having an intimate knowledge of the way in which the material being assessed was taught. (This difficulty was identified by Bloom et al themselves). This could lead to a task that was taught explicitly to students, and thus should be regarded as testing application, being assessed as involving a higher level skill such as synthesis—or *vice versa*. Another possibility is that the conveners and the assessors had different understandings of the levels in Bloom's taxonomy. All the assessors, but only a minority of the conveners, had been involved in a study group on taxonomies and assessment, so this could be the case.

The other notable finding is that several of the conveners felt that the highest levels of Bloom's taxonomy—synthesis and evaluation—were not appropriate to their module. In some cases it was clear that this was because the convener subscribed to the view that these levels would not be addressed until the final year of the degree programme. In others it seemed that it was because they felt that application was the 'core' of what computing is about and so it is appropriate to concentrate on its development in teaching and assessment.

6. A PERSPECTIVE: APPLICATION AS THE AIM

Let us take forward the idea that *application* is the aim of computer science teaching. In many disciplines, the aim of study is to develop an informed, critical perspective on the subject. For example, a history graduate would be expected not just to know lots of dates but also to be able to make critical and comparative comments on historical events, based on knowledge and theories. On the other hand, this graduate would not be expected to apply their knowledge to producing new history. Thus in such a discipline the long-term aim of study is particularly oriented towards the synthesis and evaluation levels in the taxonomy.

As noted above, a significant feature of our study of assessment in computer science modules was that the focus of assessment appeared to be at the *application* level. We might hypothesise that in disciplines such as computing the aim of study is what we might term 'higher application'. Here we are using the word *higher* in the sense that is used in terms such as 'higher criticism' or 'higher journalism'—i.e. the application informed by a critical approach to the subject, but where the criticism is not, as such, the focus of the work. In such work, the focus is at the application level in Bloom's taxonomy—yet this needs to be informed both by levels that Bloom puts below *and above*. This is illustrated in Figure 2, which contrasts with Figure 1 by adding a *higher application* capstone level.

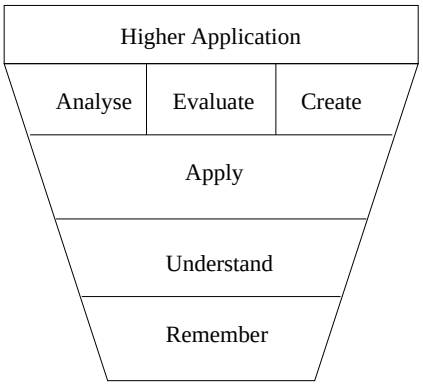

Figure 2: A suggested revised Bloom taxonomy for computing, incorporating *higher application*.

Bloom Level	Assessor 1	Assessor 2	Assessor 3	Assessor 4	Assessor 5	Mean
Knowledge	54	4	54	43	53	**42**
Comprehension	54	13	54	9	52	**37**
Application	51	43	54	5	29	**36**
Analysis	25	17	9	0	3	**11**
Synthesis	0	2	6	1	2	**2**
Evaluation	0	3	2	0	0	**1**

Table 1: Summary of assessment study: five academics rated the various assessments on the course and decided which Bloom level the material was at. The table shows how many of the assessments were rated as being at a particular level by each of the five assessors on the panel.

What other subjects might be said to have this characteristic? Clearly, subjects that are commonly compared with computing, such as engineering subjects, are of this type? Perhaps, though, this might point out similarities to more remote subjects—for example art and design subjects. Are there similarities in the way in which 'synthesis/evaluation used to improve application' is approached in those subjects? For example, peer criticism is a common approach in art and design education—is this because it is good for those subjects as such, or is it more because this is good generally for subjects that have the relationships between Bloom levels that these subjects have?

7. QUESTIONS FOR DISCUSSION

- Can a reformulation of Bloom's taxonomy provide more helpful descriptions for cognitive levels in Computer Science?
- Is the aim of computing education primarily focused on tasks that can be described as 'higher application' rather than evaluation/synthesis being the ultimate end-point of the educational process? If so, what can we learn from this?
- Should a taxonomy of learning inform the process of identifying points of reference for generic and subject-specific competences of first and second cycle graduates in Computer Science across the European Higher Education Area? If so, which taxonomy should be used?

8. REFERENCES

[1] Tuning project, tuning methodology. University of Deusto, 2004, Accessible at `http://tuning.unideusto.org/tuningeu/index.php?option=content&task=view&id=172&Itemid=205`, Accessed on June 28, 2006, 1999.

[2] Honours degree benchmark statement - computing. The Quality Assurance Agency for Higher Education, Gloucester, UK, 2000 `http://www.qaa.ac.uk/academicinfrastructure/benchmark/honours/computing.pdf`, 2000.

[3] Lorin W. Anderson and David A. Krathwohl. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Addison-Wesley, 2001.

[4] John B. Biggs and Kevin F. Collis. *Evaluating the Quality of Learning: The SOLO Taxonomy*. Academic Press, 1982.

[5] Benjamin S. Bloom et al. *Taxonomy of Education Objectives (Volume 1 : Cognitive Domain)*. McKay, New York, 1956.

[6] Raymond Lister and John Leaney. Introductory programming, criterion-referencing, and Bloom. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 143–147, New York, NY, USA, 2003. ACM Press.

[7] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education (ITICSE '06)*, pages 118–122. ACM Press, 2006.

[8] European Ministers of Education. The Bologna Declaration of 19 June 1999. Bologna, Italy, 1999. Accessible at `http://www.bologna-bergen2005.no/Docs/00-Main_doc/990719BOLOGNA_DECLARATION.PDF` accessed 4 August 2006, 1999.

[9] Terry Scott. Bloom's taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, 19(1):267–274, 2003.

Bloom Level	Questions and responses
Knowledge	**Questions:** *Is the direct learning of facts important in first year computer science, and in your module more specifically? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comments:** *'Being able to use the right words is helpful; direct learning of a formula so that they can parrot it, no.'; 'it is a language learning course, to an extent, and languages are made of facts and things.'; 'Yes, direct learning of facts is important.'* **Assessment** at this level: 6/7 modules (the other 'marginally' assessed material at this level).
Comprehension	**Questions:** *Is the ability of students to explain the course material important in your module, and in first year computer science more generally? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comment:** *'The first goal is to be able to do it, and then the second goal is to be able to explain it. Realistically I'm not sure how many of them can effectively explain what they're doing by the end, and I'm not sure how much I would let that affect my assessment. If the student does it, but does not explain it well, I would probably be reluctant to seriously penalise them for that. A proper, complete solution should include an explanation.'* **Assessment** at this level: 4/7 modules - two others 'partially'.
Application	**Questions:** *Is the application of techniques learned to new situations important in your module, and in first year computer science more generally? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comments:** *'It is essential.'; 'Yes, extremely important.' 'The more different examples they encounter the better placed they are to understand that the foundational concepts apply regardless of the context of a particular problem.'* **Assessment** at this level: 7/7 modules
Analysis	**Questions:** *Is the ability to analyse a range of information and decide which aspects of learning to apply important in your module, and in first year computer science more generally? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comments:** *'Yes, in a very constrained environment. Clearly assessed in the later assessments and in the later exam questions.'; 'That is essential. Assessed indirectly all the time. It is harder to do that explicitly.'; 'In the spreadsheets there is quite an aspect of that, but not in the more programming oriented sections.'* **Assessed** at this level: 6/7 modules.
Synthesis	**Questions:** *Is the ability to bring together diverse aspects of learning important in your module, and in first year computer science more generally? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comments:** *'No. The module sticks to a very constrained domain.'; 'To a degree, e.g. in the section on finite state machines, but it is not central. There is a small attempt to assess it.'* **Assessment** at this level: 2/7 modules (both small components)
Evaluation	**Questions:** *Is the ability to evaluate and come to judgements in the light of material learned important in your module, and in first year computer science more generally? Does this impact upon your module? If yes, do you assess this directly in your module?* **Sample comments:** *'Assessed indirectly, because some of the problems will have easier or harder ways to do them. If they have learned to identify an easier route they will do better on the exam.'; 'No, not explicitly.'; 'Looking at it, but not assessing it directly.'* **Assessment** at this level: 1/7 modules.

Table 2: Interviews with course lecturers

The Preference Matrix As A Course Design Tool

John Paxton
Montana State University
Universität Leipzig (Guest Professor)
Computer Science Department
Bozeman, MT 59717 USA
paxton@cs.montana.edu

ABSTRACT
The preference matrix is a theoretical tool based on principles of evolutionary psychology. This paper briefly introduces the theory and then describes how the preference matrix has been applied as a pedagogical design tool for an artificial intelligence course. After making a preliminary assessment of this experience, the paper concludes with several discussion questions.

Keywords
Preference Matrix, Computer Science Pedagogy, Course Evaluation

1. INTRODUCTION
The contribution of this *discussion paper* is to introduce the concept of the preference matrix and its applicability to course design. The preference matrix is a theoretical construct based on principles of evolutionary psychology. The preference matrix provides a lens with which to view course design best practices.

The paper is organized as follows. In section 2, the preference matrix construct is introduced. In section 3, it is explained how the preference matrix was used to design a specific course on the topic of artificial intelligence. Section 3 also describes a first attempt at evaluating the effectiveness of using the preference matrix to design a course. Finally, section 4 raises some discussion questions with respect to using the preference matrix as a pedagogical tool.

2. PREFERENCE MATRIX
The preference matrix is a construct of Stephen and Rachel Kaplan [7]. The Kaplans have synthesized a theory of humans as information processors that is grounded in evolutionary psychology. In order to survive successfully, an individual must be able to recognize objects in the environment (there is a grizzly bear on the edge of the meadow), make predications (the grizzly bear is coming towards me) and evaluate the consequences (this is dangerous). This is done through the use of a mental construct called a cognitive map.

Before introducing the preference matrix, it is important to examine the concept of familiarity. In Table 1 [7], the column labeled "low preference" indicates an environment that an individual does not like very well and the column labeled "high preference" indicates a preferred environment. In a "low preference" environment, a low amount of familiarity results in the individual finding the environment strange while a high amount of familiarity results in the individual finding the environment boring. In a "high preference" environment, a low amount of familiarity results in the individual finding the environment fascinating, while a high amount of familiarity results in the individual finding the environment comfortable.

To increase the likelihood that a person will spend time within an environment, it can be seen from Table 1 that the person's familiarity with the environment is less important than whether the person prefers the environment. In a preferred environment, low familiarity will catalyze the individual to engage with the environment. As a side effect of this involvement, learning will likely take place, leading the individual to function more effectively. In contrast, high familiarity with a preferred environment will foster effective functioning, but will not necessarily catalyze learning to take place.

Table 1. Familiarity matrix

	Low Preference	**High Preference**
Low Familiarity	Strange	Fascinating
High Familiarity	Boring	Comfortable

Table 2 [7] depicts different types of preferred environments. The two key dimensions that lead to an environment being preferred are (1) whether an individual can **make sens**e of the environment and (2) whether an individual can be **involved** with the environment through learning and/or exploration. Making sense and involvement can both be examined from the standpoint of time. When an environment makes sense in the present, it is considered "coherent". When an environment appears that it will make sense in the future, it is considered "legible". When an environment provides involvement in the present, it is considered "complex". And when an environment appears that it will provide involvement in the future, it is considered "mysterious". Note that the CS community is currently exploring the notion of active learning (for example, [4]). A crucial aspect of active learning is involvement.

Table 2 is called a preference matrix because the more of these four traits that are present in a given environment (coherence, legibility, complexity, mystery); the more highly preferred this environment will be.

Table 2. Preference matrix

	Makes Sense	**Involvement**
Present	Coherence	Complexity
Future	Legibility	Mystery

The preference matrix is applicable to any environment, be it natural (e.g. finding one's way in a jungle) or human designed

(e.g. a book). The remainder of this section will focus on implications that the preference matrix provides with respect to designing a course in an educational environment.

"Understanding and respecting the cognitive requirements of the intended recipient constitute probably the single most effective step one can take in improving the process of sharing knowledge". (page 195) [7] The preference matrix provides many immediate useful tips when designing a course:

- To promote making sense, new knowledge should be connected to existing knowledge. Telling a story, using an analogy and/or using a concrete example are all possible techniques for accomplishing this.
- To promote making sense, not more than 5 (plus or minus 2) major concepts should be introduced in any one session. Otherwise the short term memory capacity of the student might be overwhelmed.
- To promote involvement, it is important to develop materials that the learner cares about. Giving the learner some control over the learning process (whether it is through self-paced learning or open-ended assignments) is one way to make this happen. Games [2] also have a high involvement factor.
- To promote involvement, it is important to understand roughly the knowledge the learner brings to the course. Otherwise the learner might judge the environment to be low in "mystery" and consequently be unmotivated to learn.

3. APPLICATION

In this section, one successful pedagogical application of the preference matrix is described. In section 3.1, a brief introduction of an artificial intelligence course is given. In section 3.2, the influence of the preference matrix on this course is provided. In section 3.3, the course is assessed to determine whether the preference matrix has yielded positive benefits.

3.1 Course Overview

CS 436, Introduction to Artificial Intelligence, is a 3 credit, senior-level elective course [10]. The course is offered each fall semester and I have taught this course on 16 consecutive offerings, beginning with the fall of 1990.

The first three weeks of the course are spent introducing the required programming language, Common Lisp. The remaining 12 weeks are spent covering fundamentals of search, knowledge representation and learning in the context of practical applications.

3.2 Preference Matrix Influence

I first learned about the preference matrix when I was in graduate school during the late 1980s. When I began my career at Montana State University, the preference matrix appeared to be a good guideline to use for designing the courses that I would teach.

Although in the case of CS 436 (Introduction to Artificial Intelligence), I have taught the class 16 times and the course has gone through numerous revisions and updates, the core underlying philosophy of using the preference matrix as a course design guide has never changed.

At a very high level, the preference matrix states that a good learning environment is one where (1) a student will be able to "make sense" of the material both now and in the future and (2) a student will be "involved" with the material both now and in the future.

In each offering of the course, some of the designed features of the course that help it "make sense" to the students are

- The course objectives are clearly stated at the beginning of the course.
- A web based syllabus is designed that is simple, complete and easy to use. The syllabus is maintained on a daily basis.
- All exam questions are designed to test a student's comprehension of the course objectives. It is important to foster critical thinking on the part of the students [8].
- All programming assignments are designed to help facilitate a student's comprehension of the course objectives.
- Lecture material is presented that builds upon previous course material and what the typical student should already know. Relationships to previous material are made explicit. (For example, it is pointed out that a best first search can be implemented using a previously studied data structure: the priority queue.)
- Practical applications of lecture material (such as showing a video clip of the 2005 Mohave Desert robot race) are provided regularly.

In the Fall 2005 offering of the course, some of the designed features of the course that enhanced student "involvement" were

- One programming assignment required students to implement the k-means learning algorithm and then apply it to a problem of interest.
- One programming assignment required students to implement a Sudoku problem solver using appropriate search techniques and constraint satisfaction. The programs were evaluated based on how quickly they could solve undisclosed problems of varying difficulty.
- One programming assignment required students to implement a cribbage playing program. A class tournament then allowed the programs to play against one another. Part of the program's grade was based on its performance in the tournament. Part of the program's grade was based on the sophistication of its strategy. The cribbage assignment is a good example of the concept of "mystery". On the first day of the semester, students were told about this assignment. As the semester progressed, students knew that they must actively assemble bits and pieces of the conceptual understanding necessary to succeed on the cribbage assignment.
- During lecture, all students were called on in a systematic fashion to answer questions. Students knew that their answers would not affect their grade. Some lectures were devoted towards philosophical discussions. Dynamic interaction of all forms is an

important mechanism for fostering "involvement" [11].

Although the programming assignments change on every offering, I find that offering open-ended assignments that tap into students' interests is a very effective way to "involve" students with the course. For example, during the Fall of 2005, a Sudoku craze was sweeping campus and many of the students would work a Sudoku puzzle in the campus newspaper on a daily basis. Students were excited to have the opportunity to write a computer program to solve these problems and were surprised at how quickly a well-written program found a solution. Their intrinsic interest in the problem caused them to develop far more sophisticated solutions than what the assignment minimally required.

3.3 Results

In order to conduct an initial assessment regarding the effectiveness of using the preference matrix as a course design tool, I have examined four semesters worth of evaluation data from Fall 2004, Spring 2005, Fall 2005 and Spring 2006. During these four semesters, 19 senior level courses were offered. Two of these 19 senior level offerings were CS 436. Other instructors who do not use the preference matrix as a design guide taught the other 17 offerings. Table 3 shows the evaluation questions that students were given during the last week of the semester before finals week.

Table 3. Evaluation questions

Question	Text
Q1	How does this course compare with similar technical courses?
Q2	What is your level of interest in taking an advanced course?
Q3	Did you find this course challenging?
Q4	Were the objectives of the course clearly stated?
Q5	Were the objectives of the course met?
Q6	How important were the lectures?
Q7	How important were the assignments/programs?
Q8	How important were the tests/quizzes?

Students could respond with one of five answers: 1, 2, 3, 4 or 5. A 1 indicated the most positive response, a 3 indicated a neutral (or

average) response and a 5 indicated the most negative response.

Table 4 shows the results of the evaluation. The first column shows the question being evaluated. The second column shows the mean response for all senior level courses, excluding CS 436. There were 225 responses in this category. The third column shows the mean response for the two offerings of CS 436 that occurred during the four semester evaluation period. There were 33 responses in this category. The fourth category shows the standard deviation for all of the data collected to give the reader a sense of the dispersion. Finally, the fifth column shows the percent of improvement that the third column showed over the second column. To make a mean of 1.0 correspond to a 100% improvement, the percent improvement was computed as follows. Let x be the number from column 2, let y be the corresponding number from column 3 and let z be the percent improvement. Then $z = 1 - ((y - 1.0) / (x - 1.0))$.

As can be seen from Table 4, using the preference matrix as a course design instrument appears to improve the course significantly. For all eight of the questions, there was at least a 10% improvement and for five of the eight questions, there was greater than a 50% improvement. It is also interesting to note that students found CS 436 to be 43% more challenging than other senior level computer science courses. Thus the high evaluations are not due to the course being an easy one.

Table 4. Evaluation results

Question	Non CS-436 Mean	CS-436 Mean	σ	Percent Improvement
Q1	2.19	1.58	0.96	51%
Q2	2.53	1.67	1.29	56%
Q3	1.92	1.52	0.84	43%
Q4	1.83	1.24	0.95	71%
Q5	1.93	1.33	0.91	65%
Q6	1.79	1.52	0.99	34%
Q7	1.82	1.33	0.91	60%
Q8	2.04	1.94	0.98	10%

The three largest improvements are all related to preference matrix factors. Question 4 (were the objectives clearly stated? - a 71% improvement) is a result of helping the students to "make sense" of the course by telling them exactly and repeatedly what concepts they are supposed to incorporate into their cognitive maps. Question 5 (were the objectives met? – a 65% improvement) is a result of helping the students to "make sense" of the course concepts by explaining the concepts in such a way that students can develop a deeper and richer understanding of the concepts. Question 7 (how important were the assignments/programs – a 60% improvement) is a result of choosing programming assignments that get the students "involved". Some of the assignments are open-ended, allowing a student to explore his or her interests. Other assignments are games or puzzles (such as Sudoku), that tap into students' current interests.

It should be noted that the study reported here is an initial one. Although the results are encouraging, there are many factors in addition to the use of the preference matrix as a course design tool that could be influencing the result. Some of these factors include differing courses, differing instructors, and differing sets of students. Isolating these variables is very challenging in practice. Further thought regarding how to conduct more convincing studies is needed.

4. DISCUSSION

In this section, four discussion questions are raised. Based on comments from the reviewers, the second half of the allotted presentation time at the conference was spent facilitating a discussion on the question raised in section 4.2.

4.1 Limitations

The preference matrix is a pedagogical tool grounded in the field of evolutionary psychology. However, no tool is without

its drawbacks. As computer science teachers interested in pedagogy, we strive to be researchers as opposed to students or amateurs [9]. This requires that we examine both the strengths and the weaknesses of any given tool.

Discussion Question: What are the limitations of using the preference matrix as a pedagogical tool?

4.2 Appropriate Research Methodology

The results in section 3.3 show promise with respect to using the preference matrix as a pedagogical tool. However, it is very challenging to isolate and measure individual factors in educational studies.

Discussion Question: What are appropriate research methodologies for measuring the impact of the preference matrix in a convincing manner?

4.3 Visiting Professor Course Assessment

During Winter Semester 2006-2007, I have received a senior lecturing Fulbright Award [5] to develop and offer two courses at The University of Leipzig in Leipzig, Germany. At the department's request, one course will cover web programming topics (with an emphasis on PHP and MySQL) and the other course will cover intermediate level data structures and algorithms, motivated through ACM programming competition problems [1].

Although I will design and teach both of these courses according to fundamental tenets of the preference matrix, I am unsure how to proceed with a meaningful assessment of these courses. Challenges that must be overcome include (1) teaching the courses for the first time, (2) staying at The University of Leipzig for only one semester and (3) cultural differences.

Discussion Question: How can a course be meaningfully assessed when it is offered by a visiting professor?

4.4 Alternative Approaches

The preference matrix has served me well as a pedagogical tool.

However, other broad pedagogical approaches also exist such as Bloom's taxonomy of educational objectives [3] or even Piaget's stages of intellectual development [6].

Discussion Question: How do other pedagogical frameworks compare to the preference matrix? Are these frameworks alternative or complimentary approaches?

5. ACKNOWLEDGMENTS

My heartfelt thanks goes to Stephen Kaplan for introducing me to the preference matrix.

6. REFERENCES

[1] *ACM International Collegiate Programming Contest*. http://icpc.baylor.edu/icpc/ - Accessed July 28, 2006.

[2] Bayless, J. and Strout, S. Games as a "Flavor" of CS1. In *Proceedings of the Thirty-Seventh SIGCSE Technical Symposium on Computer Science Education*. (Houston, Texas, March 1-5, 2006). ACM Press, New York, NY, 2006, 500-504.

[3] Bloom, B. (Editor) *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company. 1956.

[4] Budd, T. An Active Learning Approach to Teaching the Data Structures Course. In *Proceedings of the Thirty-Seventh SIGCSE Technical Symposium on Computer Science Education*. (Houston, Texas, March 1-5, 2006). ACM Press, New York, NY, 2006, 143-147.

[5] *Fulbright Scholar Program*. http://www.cies.org/ Accessed July 28, 2006.

[6] Ginsburg, H. and Opper, S. *Piaget's Theory of Intellectual Development, 3rd Edition*. Prentice Hall, N.J, 1988.

[7] Kaplan, S. and Kaplan, R. *Cognition and Environment: Functioning in an Uncertain World*. Ulrich's, Ann Arbor, MI, 1983.

[8] Krishna Rao, M. Infusing Critical Thinking Skills into Content of AI Course. In *Proceedings of 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. (Monte de Caparica, Portugal, 2005). ACM Press, New York, NY, 2005, 173-177.

[9] Lister, R. Computer Science Teachers as Amateurs, Students and Researchers. In Proceedings of the 5th Baltic Sea Conference on Computing Education Research. (Koli, Finland, 2005). 2005, 3-12.

[10] Paxton, J. *CS 436, Fall 2005*. http://www.cs.montana.edu/paxton/classes/fall-2005/436/ Accessed July 27, 2006.

[11] Roberts, E. An Interactive Tutorial System for Java. In *Proceedings of the Thirty-Seventh SIGCSE Technical Symposium on Computer Science Education*. (Houston, Texas, March 1-5, 2006). ACM Press, New York, NY, 2006, 334-338.

Understanding of Informatics Systems: A Theoretical Framework Implying Levels of Competence

Peer Stechert
Didactics of Informatics and
E-learning, University of Siegen,
Germany
stechert@die.informatik.uni-siegen.de

ABSTRACT
Informatics education is concerned with how learners make sense of computational processes and devices in secondary institutions. In this article, we describe a learner-centred cognitive approach to informatics system comprehension for upper secondary education. It is part of a broader research study initiated by experience in first year informatics (CS 1) education at university level. The approach is based on object-oriented design patterns as knowledge representation carrying networked fundamental ideas of informatics and supporting the learning process. We develop a framework for informatics system comprehension consisting of three dimensions, namely learners' competencies (the learner), principles of informatics systems (Informatics) and knowledge representation (Didactics of Informatics). We conclude by describing means to achieve different levels of competence for informatics system comprehension and assign learners' activities to every level of competence.

Keywords
Education Model, Knowledge Representation, Understanding of Informatics Systems.

1. INTRODUCTION
In the knowledge society information technologies are essential in everyday life and mature citizen need informatics system comprehension. But learners have to overcome the cognitive barrier built by the complexity of networked informatics systems and their hidden principles. From a historical perspective, informatics has been a science considering an isolated computing machine. Since the 1990s, regarding computers as single machines is no longer appropriate. They are part of local networks and of the Internet. We make the need of new cognitive relations evident considering the Model Curriculum for K-12 Computer Science of the ACM that describes how to integrate the "study of computers and algorithmic processes, including their principles" into the curriculum [14]. The demand for informatics system comprehension becomes obvious in the mandatory Level II courses: "A major outcome (...) is to provide students with general knowledge about computer hardware, software, languages, networks, and their impact in the modern world" [14]. Accordingly, we define the terms informatics system and informatics system comprehension:

Definition 1: "An informatics system is the specific combination of hardware, software and networking facilities needed to solve some application problem. (...) Informatics, then, is the scientific discipline addressing construction and design of informatics systems" [4].

Definition 2: Informatics system comprehension comprises that learners have knowledge, capabilities and skills with respect to design, construction, application and network structure of informatics systems.

The didactic challenge is to present a successful education model for informatics system comprehension that will foster networked thinking for new cognitive relations.

2. AIMS AND METHODOLOGY

2.1 Related Work
This article is part of a broader research study [10]. Our approach to understanding of informatics systems at upper secondary level is motivated by experience in CS 1 education. From 2003 to 2005, our institute was responsible for the CS 1 course at the University of Siegen including class and lab lessons. Object-oriented modelling was the basis of the course and object-oriented design patterns were introduced, i.e., Composite, Singleton, Observer, Factory Method, Proxy, Model-View-Controller architecture. The work is also based on Schneider's work concerning object-oriented patterns in lower secondary education [9]. Schneider recommends the design patterns Composite, Observer, Interpreter and State [6] and formulates criteria for the selection of design patterns. Layered architectures and design patterns can be connected for the understanding of informatics systems [10]. Our approach to understanding of informatics systems combines educational research on fundamental ideas of informatics [12] and object-oriented design patterns [6]. For this purpose the author developed a didactical classification of design patterns for informatics system comprehension [13].

2.2 Scientific Aims
In the following we outline the main research questions and hypotheses, which are expatiated in Section 3.1, 3.2, and 3.3.

Research Question 1: To understand informatics systems, which characteristics of informatics systems have to be investigated and to which extent?
Hypothesis 1: There are three characteristics of informatics systems to be investigated: external behaviour, internal structure, and specific qualities (implementation details) [4].
Research Question 2: To understand informatics systems, how can we bridge the gap between learning isolated fundamental ideas and informatics system comprehension?
Hypothesis 2: We achieve informatics system comprehension via learning networked fundamental ideas of informatics. Therefore, we need an adequate knowledge representation. Object-oriented design patterns support the learning process for informatics system comprehension because they represent experts' knowledge, they are solutions to recurring design problems and carry networked fundamental ideas of informatics.
Research Question 3: What are levels of competencies according to informatics system comprehension?
Hypothesis 3: For each characteristic in Hypothesis 1, learners' activities have to be described at each level of competence.

2.3 Research Methodology

Our research methodology is inspired by successful researchers, who investigate their theories in practice, i.e., intervention by performing field studies. Prominent examples are Dagiene presenting the activities of Young Programmer's School in Lithuania [5], Hubwieser and Broy introducing a new informatics curriculum, which was "being tested in Bavaria currently" [7], as well as Schulte and Niere who present teaching strategies for secondary schools and evaluate them with teachers [11]. Such research methodology permits critically reflecting the research results.

Definition 3: We call research including field studies *Intervention Based Didactics of Informatics*.

In secondary education, there is the following situation specific to informatics education: one school seldom has enough courses in parallel to enable researchers to design studies involving experimental and control groups. Furthermore, informatics education depends on the school where it takes place, e.g., because of the informatics infrastructure. We apply case study research in the Intervention Based Didactics of Informatics, because it adds knowledge and insights regarding a phenomenon or problem identified by the researcher [2].

We also include the Didactic System [3] in the research methodology. It formalizes the learning process and is a "mapping of a didactic concept to a learning supporting series of informatics modules" [3] to make the learner active. It consists of a) knowledge structures, b) exercise classes, c) exploration modules. Prerequisites of learners, methods, and concepts are represented as nodes in a graph, i.e., a knowledge structure. Exercise classes help defining levels of competencies. Construction, description and use of learning aids like exploration modules have to be integrated into our research methodology. Our research methodology consists of six phases.

Phase 1. Analysis of the research field. We analyse the demands of learners to specify the scientific aims. Studying existing literature offers reasons why the demands are not satisfied. Finally, we analyse Informatics and Didactics of Informatics with respect to possible solutions.
Phase 2. Hypotheses. Identification of research questions demands for coarse-grained hypotheses to improve informatics system comprehension. Result of this phase is a small set of fine-grained hypotheses permitting promising research and curricula intervention.
Phase 3. Development of an education model. The hypotheses have to be combined with common theories of Informatics, Didactics of Informatics and learning theories. We use the theories to construct a learner-centred theoretical framework for informatics system comprehension and derive an education model, which forms the theoretical basis for the learning process in upper secondary education.
Phase 4. Intervention Based Didactics of Informatics. The education model for informatics system comprehension has to be implemented at upper secondary level. We apply a case-based research methodology. It offers results with respect to acceptability by learners and general feasibility. Different learning phases demand for learners' activities and new learning aids, which have to be described and developed. Therefore we consider the Didactic System. The application of the education model including the use of learning software can be implemented by the researcher and together with student teachers. The effect of being researcher and teacher in one person has to be discussed: quantitative studies can suffer from such a constellation, but we aim at qualitative results.
Phase 5. Evaluation. The learners have to take an examination. The results offer feedback concerning general feasibility. Additionally, for the evaluation of the acceptance through the learners, a written questioning is carried out and we interview the teachers of the informatics courses. The results are used for the evaluation of the Didactic System and the hypotheses. Exercise classes are validated and competencies for informatics system comprehension are formulated as a contribution to standards of informatics education.
Phase 6. Feedback. Finally, theory and learning aids will be refined. There will also be further reflection on and contribution to theory. The objective is to create workshops for teachers.

In the remainder of this article, we describe the theoretical framework and derive the education model.

3. THEORETICAL FRAMEWORK

3.1 Principles of Informatics Systems

The first dimension corresponds to the science Informatics. With respect to informatics systems, there are three characteristics [4] to be considered in a holistic way:

External Behaviour. The external behaviour of an informatics system can be investigated by informatics experiments. Experiments can apply a concrete informatics system, e.g., running a small program (or a sequence of slightly different programs) that implements and illustrates fundamental ideas of informatics by its behaviour. Learners describe the results, e.g. by functional models and use case diagrams. Animations of behaviour can be provided by learning software. Fundamental ideas, necessary concepts and design problems are identified.
Internal Structure. In general, the internal structure is only known by developers but not by users. It can be investigated rather through analysis of the components than through experiments. Learners apply different diagrams, e.g., class, object, state, and sequence diagrams to visualize the internal structure. It is necessary to consider dynamic and static representations. Design patterns as solutions to previously identified problems can be applied and connected.
Specific Qualities. They can be understood with the aid of a specification, which can serve a concrete realization (implementation details). Implementation of selected aspects of informatics systems has to be done. Learners must know programming concepts and they need to connect them. Studies show [8] that learners often fail to implement programs even if they know essential programming concepts, because they do not see the whole picture. They fail to network the concepts. It is essential to connect the implementation to results describing the internal structure of an informatics system.

These characteristics of an informatics system lead to Hypothesis 1. As mentioned above, investigation of the external behaviour has to be done in informatics experiments:

Definition 4: An informatics experiment comprises describing question and hypotheses about expected behaviour, preparation of the (technical) environment, realization / execution of the experiment, and refinement of hypotheses if needed. It is contrary to trial-and-error approaches.

The educational value of learning objectives is essential. We choose fundamental ideas of informatics as principles of informatics systems, because they provide objective criteria. A concept of informatics is called fundamental idea if it fulfils the following criteria: 1) it is observable in different areas of informatics; 2) it may be demonstrated and taught on every intellectual level; 3) it can be observed in the historical development of informatics and will be relevant in the longer term; 4) it is

related to everyday language and thinking [12]. Fundamental ideas are consensus with regard to their educational value. According to the chain of reasoning in the motivation, learning isolated fundamental ideas has not been the key to informatics system comprehension. Therefore, we suggest focusing on networked fundamental ideas (Hypothesis 2). To network fundamental ideas we need an adequate knowledge representation.

3.2 Knowledge Representation

The second dimension deals with knowledge representations, which make experts' knowledge available for learners. This is necessary to analyze situations and to solve problems. Artificial intelligence uses knowledge representations as formalisms representing rules and facts about a situation or a problem. Concepts and relations between concepts are of interest, which is also the focus of Didactics of Informatics. Examples are virtual machines and models of layers, which are helpful in advanced courses at higher education [10]. Former approaches applying block diagrams fail describing the network aspect of informatics systems, i.e., they describe isolated computing machines. Object-oriented design patterns are solutions to recurring design problems. Applying those sets emphasis on modularization and interfaces, which are essential for informatics systems. Most studies of software design behaviour "relied on one characteristic of designing, the enactment of problem solving skill" [8]. So, we conclude that design patterns are an adequate knowledge representation, because they carry networked fundamental ideas, visualize design heuristics and are solutions to design problems, i.e., we want to improve the learning process towards informatics system comprehension by offering patterns

1. to classify the behaviour of a system,
2. to structure the functioning of a part of a system,
3. to network fundamental ideas inherent in patterns,
4. to network design patterns as (parts of) informatics systems (pattern language),
5. to represent essential principles and heuristics, because they are identified hot-spots of the system, where specific problems and changes of the system occur.

We integrate design patterns into the learning process for a better understanding of informatics systems; not to qualify software engineers. For the learning process, we need a quality factor, e.g. fundamental ideas of informatics. Schwill identified them in the software engineering process [12]. So, they are consistent with Definition 1, because of a strong connection to construction and design of informatics systems. Design patterns represent networked software systems, but they also represent fundamental ideas that also occur in the context of hardware facilities and non-technical issues. They are observable in different areas of informatics. Schneider has shown that design patterns can be learned at different levels of complexity [9].

3.3 Learners' Competencies

The third dimension describes cognitive states, i.e., levels of informatics system comprehension. The levels can be described by exercise classes of the Didactic System [3] and general problems the learner is able to solve. We apply Bloom's Revised Taxonomy [1]. It distinguishes between 1) factual, conceptual, procedural, metacognitive knowledge, and 2) six succeeding cognitive processes, i.e., remember, understand, apply, analyze, evaluate, create. To every cognitive process we assign exercise classes. The characteristics of informatics systems imply the competencies for informatics system comprehension (Figure 1).

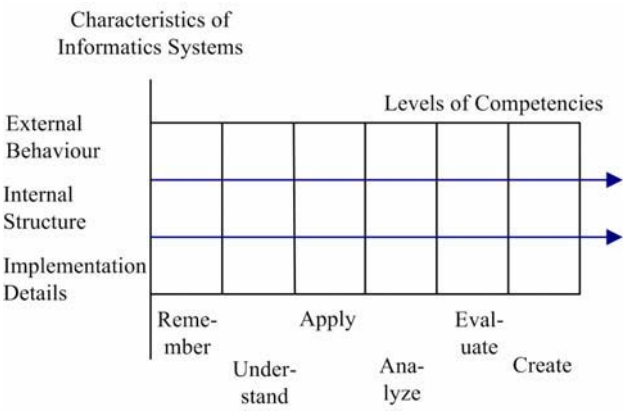

Figure 1. Levels of competencies to be achieved for informatics systems comprehension

The consideration leads to Hypothesis 3.

4. EDUCATION MODEL

A learner-centred classification of design patterns for understanding of informatics systems based on networked fundamental ideas has been done in [13]. For informatics system comprehension, we divide the learning process into three phases (S_i):

S_1: Understanding of essential aspects of external behaviour of the informatics system,

S_2: Understanding of essential aspects of the internal structure of the informatics system that are based on fundamental ideas,

S_3: Understanding of selected qualities from the set of qualities given by the specification of the informatics system (implementation details).

The three phases recur at every level of investigation and form a picture of the whole informatics system. For the learning process towards access control, we exemplarily describe learners' activities according to the levels of Bloom's Revised Taxonomy and apply Proxy, which is a structural design pattern based on object aggregation. It describes a placeholder to control access to another object [6]. The exercise classes are based on results of [3], but they have to be proved according to their effectiveness towards informatics system comprehension. It is important to mention that the education model needs a real-life scenario. e.g., music shop, and the combination of design patterns. An example for a sub-objective of S_1 is

$S_{1,1}$: Understanding of the fundamental idea access control for a list data structure with Proxy design pattern.

S_1 has to be realized by investigating input and output, which is done in informatics experiments. Different prepared programs can show unexpected behaviour, e.g., faults. Learning software showing real-life examples of fundamental ideas is adequate to explore the behaviour. Advanced learners apply functional modelling techniques and case diagrams to describe the behaviour of the informatics system. Exercise classes for $S_{1,1}$ are:

Remember: Questions about the intended behaviour of the informatics system, e.g., a small program "music shop" counting access to songs stored in a list data structure.
Understand: Learners describe the value of access control and answer understanding questions after dealing with an animation of the concept with a strong connection to real-life experiences.
Apply: Learners apply their knowledge, specify the general behaviour of a program and arrange experiments with the program implementing different aspects of the music shop.

Analyze: Learners discuss possible underpinning fundamental ideas, which permits understanding the organizational structure. They cope with unexpected behaviour by distinguishing between faults and correct behaviour.
Evaluate: Experiments with slightly changed programs, e.g., to show variations of access control, imply transformation of knowledge. That means, experiences of previous informatics experiments can be combined with unknown behaviour.
Create: -

Examples for sub-objectives of S_2 are:

$S_{2,1}$: Understanding of access control by designing an object-oriented model of a list and Proxy as validated in $S_{1,1}$,

$S_{2,2}$: Understanding of interfaces and inheritance by applying the Proxy design pattern to Composite design pattern.

For S_2, a documentation of the system, different kinds of modelling (data, functional) and diagrams (class, object, sequence, state diagram) should be used. Exercise classes for $S_{2,1}$ are:

Remember: Questions about involved classes, objects, access control, and problem solving strategies. To be a placeholder of a list, Proxy needs the same interface, which is realized by inheriting from the same abstract class. Such object-oriented concepts belong to the previous knowledge of the learners.
Understand: Understanding questions according to the relation between Proxy and the list. The learners have to eliminate irrelevant relations between classes.
Apply: Arranging the correct relations between the participating classes in a class diagram within a learning environment, e.g., Proxy needs a relation to a list to call it after counting the accesses. Learners construct a state diagram of access control.
Analyze: Role-playing typical scenarios results in understanding the internal processes, e.g., accessing a song.
Evaluate: Learners modify given models and transform them to another representation or another context.
Create: Learners construct an object-oriented model for a music shop, which only allows a certain number of accesses.

An example for a sub-objective of S_3 is

$S_{3,1}$: Understanding of programming parts of the designed object-oriented model including Proxy design pattern.

There has been much research on programming competencies, which we will not explain for every level. It is essential for learners to connect programming concepts and to see the whole informatics system while programming an isolated line of code. So, the exercises of S_3 should include links to S_1 and S_2. In the example, counting access applying a loop and controlling access by programming to a common interface of Proxy and a list have to be connected. We are developing the learning software "Pattern Park" to connect the described phases [13].

5. OPEN QUESTIONS AND DISCUSSION

We present our approach to informatics system comprehension and argue for a design, intervention, evaluation cycle of curriculum development in secondary schools, which has to be discussed. Furthermore, we construct a theoretical framework for informatics system comprehension that demands for regarding different characteristics of informatics systems, choosing valuable informatics content (fundamental ideas) and learning it in a networked way within design patterns as knowledge representation. In combination with the Didactic System the framework permits developing hypotheses for the research questions. It has to be discussed whether and how the research questions have to be refined for scientific investigation of informatics system comprehension at upper secondary level. In particular, we assign learners' activities to the levels of Bloom's Revised Taxonomy. It has to be discussed how to foster networked thinking, how to engage students at the upper levels of the taxonomy, and how to assess their understanding of informatics systems.

6. REFERENCES

[1] Anderson, L. W. and Krathwohl, D. R. (eds.). *A taxonomy for learning, teaching and assessing: A revision of Bloom's Taxonomy of educational objectives.* Addison Wesley Longman, New York, 2001.

[2] Bassey, M. *Case study research in educational settings*. Chapter 7: Methods of enquiry and the conduct of case study research, UK: Open University Press, 1999.

[3] Brinda, T. and Schubert, S. Didactic System for Object-oriented Modelling. In: Watson, D. and Andersen, J. (eds.): *Networking the Learner. Computers in Education.* Kluwer Academic Publisher, Boston, 2002, 473-482.

[4] Claus, V. and Schwill, A. *Duden Informatik*. 4. Auflage, Duden Verlag, Mannheim, 2006.

[5] Dagiene, V. Programming-Based Solutions of Problems In Informatics Curricula. In *IFIP WG 3.1 and 3.5 Open Conference "Communications and Networking in Education: Learning in a Networked Society.* 1999, 88-94.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[7] Hubwieser, P. and Broy, M. Educating Surfers Or Craftsmen: Introducing An ICT Curriculum For The 21st Century. In *IFIP WG 3.1 and 3.5 Open Conference "Communications and Networking in Education: Learning in a Networked Society. 1999*, 163-177.

[8] McCracken, W. M. Research on Learning to Design Software. In: Fincher, S. and Petre, M. (eds.) *Computer Science Education Research*, Taylor & Francis, London, 2004, 155-174.

[9] Schneider, M. Design pattern, a topic of the new mandatory subject informatics? In van Weert, T. and Munro, R. (eds.): *Informatics and The Digital Society: Social, Ethical and Cognitive Issues*, Kluwer 2003, 157-171.

[10] Schubert, S. From Didactic Systems to Educational Standards. In *Proceedings of the 8th IFIP World Conference on Computers in Education.* Cape Town, South Africa, 2005, Documents/397.pdf.

[11] Schulte, C. and Niere, J. Thinking in Object Structures: Teaching Modelling in Secondary Schools. In: *Proceedings of the ECOOP Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts*, Spain, 2002.

[12] Schwill, A. Computer science education based on fundamental ideas. In Passey, D. and Samways, B. (eds.) *Information Technology - Supporting change through teacher education*. Chapman Hall, 1997, 285-291.

[13] Stechert, P. Informatics System Comprehension - A learner-centred cognitive approach to networked thinking. In: Education and Information Technologies, ISSN: 1573-7608, Springer Netherlands, 2006.

[14] Tucker, A. (ed.). *A model curriculum for K-12 computer science: Final report of the ACM K-12 task force curriculum committee*, ACM, New York, 2003.

"I Think It's Better if Those Who Know the Area Decide About It"

A Pilot Study Concerning Power in CS Student Project Groups

Mattias Wiggberg
Department of Information Technology, UpCERG
Uppsala University
Box 337, S-751 05, Uppsala, Sweden
mattias.wiggberg@it.uu.se

ABSTRACT

In the light of an emerging interest in project work within the CS curricula, research about projects is gaining importance. In the current work I investigate a full semester CS project course, where focus is on how power is distributed within a group of students. CS skills are shown to be a contributing factor when it comes to power. Using a phenomenographic research approach, a way of researching some aspects on power within CS projects groups is demonstrated. Finally, three qualitatively different ways of experiencing CS skills of other students, and thus power, are revealed.

1. INTRODUCTION

In this work I examine a method to study students' understanding of CS skills, later on referred to as perceived competence, and the relationship between perceived competence and power within student project groups in CS. This pilot study focuses in particular on the role of perceived competence as a power base. In the light of the emerging research on collaborative student projects in higher education in CS, for example Barker [1] and Berglund [3], extended knowledge about social skills and their interactions are shown to be desirable. This is especially true when one of the motivations for doing projects within the CS curriculum is the linking of social and technical skills. The collaborative teaching form also offers a deeper understanding of the context in where the technology is supposed to be implemented [11].

Open Ended Group Projects (OEGP) [10], like the one explored in this study, are becoming increasingly popular. They offer a learning environment where social components are important. Authority, roles and hierarchy all have an influence on the learning process [2]. Therefore, it is important and relevant to question the role and impact of power in these student projects. This study does not just take a step in the direction of knowledge about power interaction within student projects, but also develops a method for doing such research.

OEGP has a component of social interaction [10] where the current research project has a contribution to make. The literature within the field of CS education research consists of only a few studies remotely close to this subject (see for example Barker [1] and Berglund [3] for a note-worthy exception) which also stresses the need for further research.

The point of departure in the current study is a full semester project course for final year IT engineering students in collaborative project work, aiming at designing and implementing an advanced robotic system. The technical level of the resulting robotic system is specified in advance, but the design of the final system depends on the students' own choices.

[0]The title is a citation from the interviews by Lukas.

1.1 Research Questions

The main research question of this pilot study is how the students become aware of competence of other students in the CS project group. To put it more precise, *what makes CS students experience fellow students as being competent within the subject area?* Thus, the phenomenon investigated is the manner in which CS students experience their fellow students as competent. This question is to be seen as an initial probe for the feasibility and relevance of research in the more general area of power within student project groups.

2. RELATED WORK

2.1 Definitions of Power

A classic definition of power, which is the one used in the reported study, is that of Dahl: "A has power over B to the extent that he can get B to do something that B would not otherwise do." (p. 203) [7]. Moreover, Provan [16] argues that power can be divided in potential and enacted power. Potential power, on one hand, is for instance based on position, formal authority and membership in groups having control over key decisions. On the other hand, enacted power is the "demonstrated ability to affect organizational outcomes" (p. 7). This means that Provan makes a distinction between shown, or demonstrated, power and power that comes with a position in a certain place.

Furthermore, power can have many faces. French & Raven identify five different forms of power; coercive power (the power to force someone to do something), reward power (the ability to ask people to do things in exchange for something they want), legitimate power (power connected to a role), referent power (the power given by someone who adores you and wants to be like you) and finally, expert power (when someone has knowledge and skills that someone else requires) [17].

2.2 Project Work Among Students

The issue of project work as an educational setting in engineering and CS has been investigated in several papers, for instance Brown and Dobbie [4], Coppit and Haddox-Schatz [6], Newman et al. [15] and Leeper [13].

Seat and Lord [18] emphasize the importance of practicing interpersonal skills like communication and teaming. They refer a program for teaching interaction skills to engineers with the aim of increasing the efficiency of their technical skills. The approach for teaching those soft skills was to let the students adopt a simple set of general principles and apply them to their own

context. From there, the students could play and interact in supervised groups with the possibility of getting feedback.

The students' motives for learning within a project environment are elucidated by Berglund [3] where the social dimension, academic achievement and project skills are identified. Berglund has investigated and reported on the control structure of CS teams in a very similar social context to the current study. Tensions or contradictions in the groups have also been identified and exposed as a part of the social game within the group.

Barker [1] sheds light on how unclear aims in projects have a negative influence on the learning environment and pedagogic outcome of the project model. Even though performed in another social context, Barker presents findings worth considering. One of the more noticeable results from that study is the unawareness of the effects of knowledge asymmetry, which happens when one group member is more skilled in a topic than the others are. Knowledge asymmetry can be used for peer tutoring, a beneficial situation for both parties. But, when students select their own roles in the group, they often tend to choose task where they already are more skilled in and by that lose the major impact of the peer tutoring in collaborative work. This also implies that in a group allocating tasks themselves, improved learning does not automatically follow. Barker also argues that only when group processes are made explicit, can activities lead to enhanced learning.

3. PHENOMENOGRAPHY

In order to explore the complex question of why somebody has a stronger position in the CS project group, I take a phenomenographic approach. Phenomenography is a research framework for revealing the qualitatively different ways in which people experience a phenomenon. The approach is a second order research perspective that tells something about other peoples' experience of the world. The opposite, a first order research perspective, makes statements about the world [14]. Thus, in order to learn about how students experience competence of others, phenomenography is an appropriate approach.

A phenomenon can be experienced in many different ways. The rationale behind phenomenography is to find and describe the outcome space which consists of the different ways of experiencing the particular phenomenon [14]. An important characteristic of a valid phenomenographic outcome space is the relationships between the categories. Cope [5] describes this:

> "One of the consistent findings of phenomenographic studies is that a group of individuals will experience the same phenomenon in a limited number of distinctly different ways. Importantly the different experiences have been found to be related hierarchically based on logical inclusiveness and increased level of understanding." (p. 68) [5]

The concept of awareness of a phenomenon can be understood as its meaning and its parts and their relationship [14]. Together these two aspects create a whole. Berglund [3] has an example of this:

> "A coin of one euro can serve as an illustration: To get a full picture of such a coin, both the meaning (a currency in many European countries; that is, a legal tender) and its shape (round, consisting of two different metals) must be known." (p. 40-41) [3]

4. THE STUDY

The empirical data was collected from a CS project course in the final year of the IT engineering program at the Department of Information Technology, Uppsala University. The course duration is one semester. The students together carry out a task of designing and building a power line inspection robot [9]. The project course usually involves two 12 person teams with support of 2-4 teachers. An earlier instance of this course is described in Daniels and Asplund [8].

The student cohort was analyzed by study background, stated interests and project roles. From that analyze, eight students representing a great variety concerning those variables where selected. Those eight students where then interviewed. The interviews were transcribed verbatim. An iterative process of identifying and categorizing the experiences followed, where sorting and resorting piles of excerpts was a major activity. Finally, an unambiguous distribution of excerpts in categories where found and thus the iteration ended.

5. EMPIRICAL RESULTS

The study shows two results. Firstly, perceived competence, presumed or demonstrated, leads to increased influence. Secondly, three different ways of experiencing competence can be found in the project group: presumed skills, earlier demonstrated skills and demonstrated skills.

5.1 Perceived Competence as Contribution to Influence

All interviews conducted indicates that perceived competence is a contributing factor when it comes to influence within the student project. One typical example is the excerpt from Emil [1].

> **Interviewer**: Are there some [students] whose opinions get more attention?
> **Emil**: Yes, those who have competence. It feels like William and Lukas have most.
> **Interviewer**: And people listen to him?
> **Emil**: Yes.

This result is expected and in line with French & Raven's discussion about expert power and increased influence; by expressing yourself as competent you increase your influence.

5.2 Experiencing Competence

Three qualitatively different ways of experiencing competence of fellow students within the current student project have been identified. These three categories constitute the phenomenographic outcome space. Since the three presented categories build on each other and each of them contributes with a qualitatively difference, they are connected.

5.2.1 Category One: Presumed Skills

This category holds expressions that support a presumption of competence. The expressed thoughts about someone's competence are not founded upon any evidence thereof, but rather on presumptions. The following excerpt from Emil illustrates the category.

> **Interviewer**: And what is it that makes people listen to them?
> **Emil**: That they seem to know what they talk about.

[1]To preserve the anonymity of the students, their names in all excerpts are replaced by fake ones.

Another student, William, emphasizes this when he talks about how he was appointed a certain task.

> **Interviewer**: Why do you think Oskar appointed you?
> **William**: Because he thought I knew what I was talking about.

5.2.2 *Category Two: Earlier Demonstrated Skills*

This category holds expressions of skills demonstrated in earlier settings. Experiences of someone's abilities to solve problems, not within but close, to the current project focus are articulated. Thus, presumptions about someone as competent are based on evidence, but not from the same area as the project deals with. Again Emil helps us with an example of this category.

> **Interviewer**: Then competence is something one takes into account?
> **Emil**: Yes, I definitely think so. It's not like one puts someone that doesn't, sort of isn't used to, having responsibility for a server to have it. One rather takes someone that has it, already have responsibility and experience from before.

5.2.3 *Category Three: Demonstrated Skills*

This category describes that someone's acting during the current project work constitutes the bases for fellow student's interpretation of the competence of him/her. Showed skills within the present project are interpreted as evidence of competence. The category implies that the subject is presumed to be competent, but now with evidence from the current project. The difference is that the evidence for the presumed competence is from the project setting where the competence is needed. Let us hear how Erik explains the core feature of this category.

> **Interviewer**: Did he get responsibility at the start, [...]. To decide this much?
> **Erik**: I don't think he decided all that much in the beginning, it sort of grew. He has proved himself competent several times. And the more he come through as competent, that he made the right decisions, the more we others allowed him.

5.3 Discussion

The initial result that perceived competence contributes to influence in student projects in CS is emphasized in all of the interviews performed. This is therefore the starting point for the data driven phenomenographic analysis that leads to the second result.

The result concerning how students experience the competence of their fellow students is summarized in table 1. The table differentiates between the meaning of the categories and the relationship between them.

Focusing on the differences, the inclusiveness between the categories can be elaborated. The first category and the second category have its main difference in the expressions of skills in earlier CS projects.

Let us listen to a continuation of the last excerpt from category one, where William gives an example of the inclusiveness and the difference.

> **Interviewer**: Why do you think Oskar appointed you?
> **William**: Because he thought I knew what I was talking about.
> **Interviewer**: There was thus a presumption...
> **William**: Yes, we have also worked together before.
> **Interviewer**: You know each other?
> **William**: Yes, everyone in the project has more or less worked together before except Lukas and Alexander.

In the next category, demonstrated skills, the qualitative difference is that the demonstrated skills are from within the current project. However, still the experience of competence is based on evidence from earlier projects and then increased with experiences of skill from the current one. Emil, who first expresses an earlier demonstrated competence and later also states experiences from the current setting as indications for competence, describes the differences for us.

> **Interviewer**: Did he come in with this responsibility [..] What made you presume, to understand, to know, that Lukas mastered it?
> **Emil**: At the start it was just because he studied at the Electronic engineering study programme. And the... it has become clear that he is very competent. That he does good things.

Thus, the categories are getting more and more detailed with respect to their requirements about skills relevant to the current project.

6. CONCLUSION

Two important conclusions can be drawn from the study. First, perceived competence contributes to personal influence in the student project groups. Second, three qualitatively different ways of experiencing competence among other students have been identified.

The first result about perceived competence has obvious similarities with results from other studies regarding the value of competence. For instance Grant et al. [12] concludes in an article regarding the importance of technical competence to project managers that: "A majority of respondents in the sample, regardless of personal or situational factors indicated technical competence is extremely important or absolutely essential" (p. 17). Barker and Garvin-Doxas [2] emphasize that status is something you earn in the classroom by giving evidence of skills: "status is informally accorded to those who display their ability to write 'elegant programs', display ability to reason well [...] or provide other needed information" (p. 16).

How well these different studies are comparable with the current study is of course an issue for discussion. Being studies of another context (the defence acquisition) or other settings (classrooms), they still could support the current finding.

The second result is in accordance with the work about potential and enacted power of Provan [16], someone's possibility to control project decisions is based on competence and not formal positions. Thus, competence as a source for power in the current study can be identified as leading to enacted power.

6.1 Open Questions

There is an emerging focus on team work in the CS curriculum. Despite this, the research that is performed on human power and the effects of power on the learning outcome in students' teams in CS are still limited. With respect to those circumstances and

Category	Meaning	Characteristics
One	Presumed skills	Expressions that support a presumption of competence.
Two	Earlier demonstrated skills	Expressions of abilities to solve problems not within, but close to, the current project. Presumptions about someone as competent are based on evidence, but not from the same area as the project deals with.
Three	Demonstrated skills	Someone's actions during the current project work constitute the basis for fellow student's perception of his/her competence. Gradually shown skills within the particular field of application are interpreted as evidence of competence. The category implies that the student is presumed to be competent, but now with evidence.

Table 1: Categories of description of what makes CS students experience fellow students as being competent within the subject area.

the indications derived from the current study that power is an influencing factor, the following initial research questions are proposed for further research;

- in what ways are influence and responsibility as well as the organization of the teams related to the learning outcome,
- are there other ways of distributing influence than perceived competence,
- is the distribution of influence in the teams related to the students' perceived competencies of CS, and
- what can be learned about influence and responsibility in order to prepare rewarding project settings?

The presented work also opens up for several relevant methodological and legitimacy questions connected to my Ph.D. work, where I am especially interested in;

- how well phenomenography can be used for investigating influence, and
- how power and social interaction research has its application within CSED?

7. REFERENCES

[1] L. J. Barker. When do group projects widen the student experience gap? In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 276–280, New York, NY, USA, 2005. ACM Press.

[2] L. J. Barker and K. Garvin-Doxas. Making Visible the Behaviors that Influence Learning Environment: A Qualitative Exploration of Computer Science Classrooms. *Computer Science Education*, 14:119–145, jun 2004.

[3] A. Berglund. *Learning computer systems in a distributed project course: The what, why, how and where*. Acta Universitatis Upsaliensis, Uppsala, Sweden, 2005.

[4] J. Brown and G. Dobbie. Software engineers aren't born in teams: Supporting team processes in software engineering project courses. In *SEEP '98: Proceedings of the 1998 International Conference on Software Engineering: Education & Practice*, page 42, Washington, DC, USA, 1998. IEEE Computer Society.

[5] C. Cope. Educationally critical aspects of the concept of an information system. *Informing Science*, 5(2):67–79, 2002.

[6] D. Coppit and J. M. Haddox-Schatz. Large team projects in software engineering courses. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 137–141, New York, NY, USA, 2005. ACM Press.

[7] R. Dahl. The concept of power. In *Behavioral-Science*, volume 2, pages 201–215, USA, 1957. General Systems Science Foundation.

[8] M. Daniels and L. Asplund. Multi-level project work; a study in collaboration. In *30th ASEE/IEE Frontiers in Education Conferences*, pages F4C–11 – F4C–13, Kansas City, MO, USA, 2000.

[9] T. Danielsson, M. Olsson, D. Ohlsson, D. Wärrmegård, M. Wiggberg, and J. Carlström. A climbing robot for autonomous inspection of live power lines. In *Proceedings of ASER06, 3rd International Workshop on Advances in Service Robotics*, Vienna, Austria, July 2006.

[10] X. Faulkner, M. Daniels, and I. Newman. Open ended group projects (OEGP): A way of including diversity in the IT curriculum. In G. Trajkovski, editor, *Diversity in information technology education: Issues and controversities*, pages 166–195. Information Science Publishing, London, 2006.

[11] B. Friedman and P. H. Kahn. Educating computer scientists: linking the social and the technical. *Commun. ACM*, 37(1):64–70, 1994.

[12] K. Grant, C. Baumgardner, and G. S. Shane. The perceived importance of technical competence to project managers in the defense acquisition community. *IEEE Transactions on Engineering Management*, pages 12–19, 1997.

[13] R. Leeper. Progressive project assignments in computer courses. In *SIGCSE '89: Proceedings of the twentieth SIGCSE technical symposium on Computer science education*, pages 88–92, New York, NY, USA, 1989. ACM Press.

[14] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Associate, Mahwah, NJ, USA, 1997.

[15] I. Newman, M. Daniels, and X. Faulkner. Open ended group projects a 'tool' for more effective teaching. In *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, pages 95–103, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[16] K. G. Provan. Recognizing, measuring, and interpreting the potential/enacted power distinction in organizational research. *Academy of Management Review*, pages 549–559, Oct. 1980.

[17] B. Raven and J. French. *The bases of social power*. Harper and Row, 1960.

[18] E. Seat and S. M. Lord. Enabling effective engineering teams: a program for teaching interaction skills. In *Frontiers in Education Conference 1998. FIE '98. 28th Annual*, volume 1, pages 246 – 251, Tempe, AZ, 1998.

Demo/Poster Papers

ALOHA - A Grading Tool for Semi-Automatic Assessment of Mass Programming Courses

Tuukka Ahoniemi
Institute of Software Systems
Tampere University of Technology
P.O.Box 553
FI-33101 Tampere, Finland
tuukka.ahoniemi@tut.fi

Tommi Reinikainen
Institute of Software Systems
Tampere University of Technology
P.O.Box 553
FI-33101 Tampere, Finland
tommi.reinikainen@tut.fi

ABSTRACT

Many mass programming courses face the problems related to using multiple graders for student assignments: achieving objectivity and consistency in grading. Grading rubrics are a possible solution to this problem. ALOHA is an online grading tool that provides rubrics that all the graders have to use. ALOHA also provides features that make the grading process, including the writing of a feedback text, more convenient for the graders and the teacher.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

1. INTRODUCTION

Novice level programming courses in Tampere University of Technology tend to be large mass courses with hundreds of students. On one hand we would like to keep our workload of these courses maintainable but on the other hand novice students require profound and personalized feedback on their programming assignments. Fully automatic assessment would easen the workload, but does not provide the important feedback on matters that cannot be automatically assessed and have a lot of focus on novice courses, like programming style [2]. As a compromise we use semi-automatic assessment [1]. This usually requires the mass course to use multiple graders for the student assignments.

The obvious aim is to achieve a consistent and objective grading between different graders. In practice, this often turns out to be quite difficult [4]. To aid courses in this as well as in providing proper feedback for the students we present an online grading tool called ALOHA.

2. PROBLEMS OF OBJECTIVE GRADING

To ensure consistency among graders, the graders need to be properly trained for the job and they are required to use marking schemes to direct their attention to the appropriate things in a student's work [5]. Becker suggests a similar approach and also defines the schemes as *rubrics* [3].

The use of rubrics increases objectivity because the idea is in splitting the assessment into smaller parts. The grader can concentrate on a single aspect of the work instead of giving a general grade for the final work. The total grade can later be derived from the grades given to the parts.

According to Habeshaw et al. [5], the only definite way to grade objectively is by using only objective tests (multiple-choice questions etc.), but doing so would not be desirable in courses where a major part of the learning is based on the student programming by himself. By receiving in-depth feedback the student can focus on improving his weaknessnes in the future. So the only way to keep this and still achieve objectivity is to divide the assessment in small enough parts with rubrics.

Earlier we used traditional rubrics in grading but unlike Becker we did not let the students see the filled rubrics. Thus all graders did not use the rubrics properly but formed the grade before even filling the rubric. The purpose of ALOHA is to provide the rubrics online in a way that each grader has to fill them correctly and in a similar way.

The fact that many of the dozens of student submissions are alike causes problems. The submissions resemble each other due to student guidance, but also because the assignments are kept moderately simple and very strict due to the nature of CS1-level teaching. Because of this alikeness among submissions, the grader easily starts to feel as if he is grading *the same work over and over again*. This in turn may tempt the grader from writing in-depth feedback.

3. GRADING WITH ALOHA

To help the grader in writing profound feedback ALOHA has a feature one could call *semi-automatic phrasing*. This means that when the grader gives a grade for a small subpart he can choose a ready-made phrase to be added for the student's feedback. This helps the grader not to have to write the same kind of feedbacks all over again for each of the student doing the same mistake. The phrases are related to a certain assignment and added beforehand by the lecturer who has set the rubric up. Still the grader has the ability to modify old and create new phrases.

Semi automatic phrasing has been a feature of which the graders have liked a lot and because of it they have felt that the tool is made to help them in the grading and not just ensuring that everyone is grading in a similar way. The tool is also a great aid for the lecturer to manage the whole grading process as he has all the gradings and final graders in one collective online place.

Figure 1 shows the grading view in ALOHA. This view presents the grading rubric. The grading is divided into hierarchy shown in the left consisting of categories (*Documentation, Dynamic tests, etc.*) which consist of subcategories (*First document, Final document, etc.*). Each of the subcategories is given a grade based on the *grading*

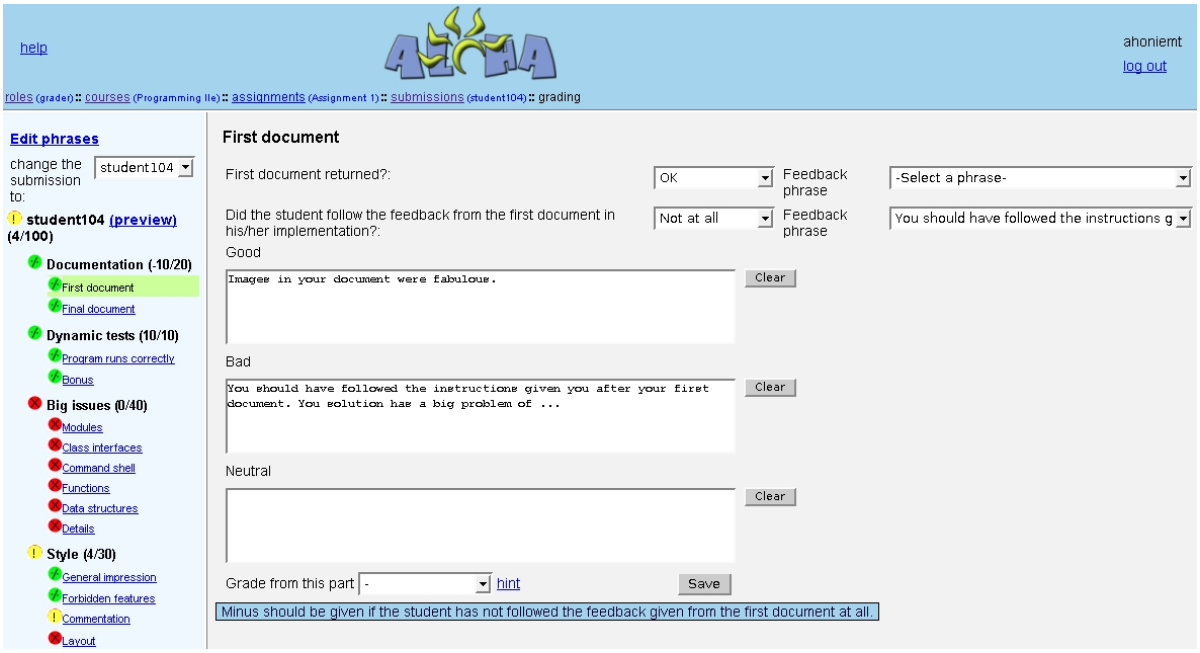

Figure 1: Screenshot of the grading view in ALOHA

items related to it (*"First document returned?"*). Each grading item has a collection of ready-made phrases to be added to the feedback of that subcategory.

The lecturer can decide on different weightings on categories. The grade limits are also customizable resulting in quite free opportunities on how to use the tool. This requires a bit more effort on setting up the tool on behalf of the lecturer, but allows the tool to be used in many different courses - some possibly even unrelated to computer sciences.

4. DISCUSSION

The tool is useful in courses where there are several submission to be graded by multiple graders but with only small amount of submissions the benefits are mostly limited to the more comfortable grading process. Of course if the grading takes several days of time the tool might help the grader to stay consistent throughout the whole process.

The objectivity aspect of ALOHA is not that useful when an assignment is graded with the scale accepted/rejected especially if almost every student should pass the assignment. The feature that calculates the actual grade is obsolete, but ALOHA can still be used to calculate possible bonus points. Nevertheless, the tool can still be used to write good feedback texts.

ALOHA is currently developed and used only in programming courses, but its usage is actually not at all limited to programming nor even computer sciences. The limiting issue is that the building of a grading schema for an assignment requires moderate programming skills. Because it does not really have to be the teacher himself who does this, the tool could be used in other disciplines as well for example to grade project works or essays.

5. CONCLUSIONS

We have introduced few problems related to using multiple graders which is common in mass courses. ALOHA was built to easen these problems concerning consistency and objectivity in grading but also to make the grading process more comfortable for the grader resulting in proper feedback for the students.

The tool has been taken in use and it seems to make the grading process more convenient for the graders and also for the teacher. To test the consistency and objectivity, we will conduct a statistical analysis between the grading distribution of the graders using ALOHA to see if there are any differences.

6. ACKNOWLEDGMENTS

We would like to thank the rest of the group implementing the tool: Päivi Aho, Jarno Keskikangas, Harri Luoma and Hanna Sillanpää.

7. REFERENCES

[1] K. Ala-Mutka and H.-M. Järvinen. Assessment process for programming assignments. *Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT'04)*, 2004.

[2] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, June 2005.

[3] K. Becker. Grading programming assignments using rubrics. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 253–253, New York, NY, USA, 2003. ACM Press.

[4] M. Daniels, A. Berglund, A. Pears, and S. Fincher. Five myths of assessment. In *ACE '04: Proceedings of the sixth conference on Australasian computing education*, pages 57–61, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[5] S. Habeshaw, G. Gibbs, and T. Habeshaw. *53 Problems With Large Classes*. Technical and Educational Services Ltd., Bristol, U.K., 1992.

Plaggie: GNU-licensed Source Code Plagiarism Detection Engine for Java Exercises

Aleksi Ahtiainen
Helsinki University of Technology
P.O.Box 5400
FI-02015 HUT Finland
aleksi.ahtiainen@iki.fi

Sami Surakka[1]
Helsinki University of Technology
P.O.Box 5400
FI-02015 HUT Finland
sami.surakka@hut.fi

Mikko Rahikainen
Helsinki University of Technology
P.O.Box 5400
FI-02015 HUT Finland
mikko.rahikainen@iki.fi

.

ABSTRACT
A source code plagiarism detection engine Plaggie is presented. It is a stand-alone Java application that can be used to check Java programming exercises. Plaggie's functionality is similar with previously published JPlag web service but unlike JPlag, Plaggie must be installed locally and its source code is open. Apparently, Plaggie is the only open-source plagiarism detection engine for Java exercises.

Keywords
Cheating, computer-assisted instruction, Java, open source, plagiarism, source code plagiarism detection engine

1. INTRODUCTION
A source code plagiarism detection engine Plaggie is presented. It is a stand-alone Java application that can be used to check Java programming exercises. Java is a common language for introductory programming courses. McCauley and Manaris [5] reported that 56% of accredited undergraduate computer science programs in the USA expected to use Java as their first language during the academic year 2002–2003.

Several source code plagiarism detection engines exist (Section 2). After the literature review in 2002, we selected JPlag [6] and MOSS [1] out of seventeen engines to be tested more thoroughly. We found JPlag as being the most promising for the needs of our laboratory. However, JPlag could not make a difference between (a) program code that was programmed by students and (b) program code that was distributed to them as a part of an exercise. This was a problem because one of our laboratory's programming courses used heavily exercises where some program code was distributed to students. Therefore, we decided to program a JPlag-like detection engine.

This shortcoming is solved in the current version of JPlag. Thus, the most important or the only contribution of Plaggie is that its source code is open (see Section 2).

JPlag or MOSS is probably the most suitable alternative for most computer science departments but Plaggie might be more suitable in the following situations: (a) A computer science department does not want to take a slightest risk that confidential information about cheating is sent outside the institution by mistake. For example, a teaching assistant might forget to remove student names or identification numbers from the submissions before sending them to a web service. (b) Someone wishes to develop Plaggie further or integrate it with other computer-assisted or computer-managed instruction software already in use. (c) Someone wishes to compare different source code plagiarism detection engines in detail using white-box testing instead of black-box testing.

2. RELATED WORK
For brevity, only one general publication about cheating and one review about source code plagiarism detection engines are presented. Wagner's web page [13] is a general discussion about cheating in computer science education including practical advice. For example, he discussed whether hardline or compassionate strategy is better.

Lancaster and Culwin [4] compared eleven source code detection engines and recommended web-based services JPlag and MOSS. We found these two engines as most interesting as well. They classified nine out of eleven engines as local (p. 104) and wrote: "Institutions wishing to use an entirely localized detection solution, for instance to avoid issues when submitting student work to an external source, should consider downloading and installing YAP3…" (p. 114–115). Unfortunately for us, YAP3 cannot compare Java programs (p. 105).

Some detection engines compared by Lancaster and Culwin [4] are listed in Table 1. Only the engines that can be used to check Java exercises are included. We searched at the web whether the source code of an engine was available and open. The results of this search are presented in the table.

Table 1. Some properties of detection engines for Java exercises

Engine	Source code available?	Source code open?
Big Brother	Unclear	Unclear
DetectaCopias	Yes	No
JPlag	No	No
MOSS	No	No
Saxon	No	No
SIM	Yes	No

Lancaster and Culwin [4, p. 104] wrote about Big Brother's availability: "special arrangement" which probably means that the source code is not open and the program might be available via email, for example. The source codes of DetectaCopias and SIM are available at the web. However, DetectaCopias did not use tokenization [4, p. 106], and in August 2006, DetectaCopias' instructions for use and comments in the source code were in Spanish, and the source code did not include copyright or license information [10]. In October 2006, SIM did not include copyright or license information [2].

In addition to engines compared by Lancaster and Culwin, we tried but did not succeed in finding open-source engines that

[1] Surakka is the corresponding author

can be used to check Java exercises. The main results of this search are presented next.

SourceForge.net is the world's largest open source software development web site. We found three related projects from SourceForge.net: The BOSS Online Submission System, Sunlight, and Plagiarism Prevention Tools for Teachers. BOSS is a course management system originally developed for programming courses [12]. The project status is 4 – Beta and the source files are available for download [9]. The plagiarism detection engine Sherlock was integrated into BOSS in 2002 [12]. However, Sherlock is targeted at text analysis [11].

According to SourgeForge.net [8], "Sunlight will be an umbrella project for an exploration of source code similarity detection techniques, although the immediate and initial goal is to produce a C++ similarity measurement tool to allow for the detection cheating." The development status is 1 – Planning; that is, the project does not distribute any program files.

The project Plagiarism Prevention Tools for Teachers is apparently targeted at text analysis because they wrote: "…verifying the originality of the papers." The development status is 2 – Pre-Alpha and the project does not distribute any program files. [7]

3. PLAGGIE'S FUNCTIONALITY AND AVAILABILITY

Plaggie can check programs that are written in Java 1.5 also known as Java 5. Plaggie is GNU-licensed and can be downloaded from the web page of our institution [3]. Plaggie has been tested in Linux. Installing to Windows should be possible but this has not been tested. The file README_PLAGGIE includes the installation instructions, description of the algorithm, the list of features, and more.

4. DISCUSSION

When detection and handling cheating in programming exercises is considered as a whole, most work is done *after* a detection engine finds suspicious-looking student programs. A teaching assistant and/or a lecturer read the student programs, students are asked for explanations, and so on. Based on our experience and discussions with others, Plaggie is used 10–40% of time and the rest takes 60–90%. Therefore, it has a little practical meaning that web services such as JPlag and MOSS are somewhat slower than locally installed engines such as Plaggie.

It is possible that some developers of detection engines do not distribute their engines via web because this might ease cheating. However, we agree with Grune and Huntjens who wrote [2]: "We are not afraid that students would try to tune their work to the similarity tester. We reckon if they can do that they can also do the exercise." A bigger risk is that some company already selling essays and theses to students starts offering solutions to programming exercises as well. For example, ThesisExpress.com advertises: "Our system is powered by Plagiarism-Finder from Germany." It might be a hard-to-tackle cheating service if a company such as ThesisExpress.com used an auction service like Rent A Coder (www.rentacoder.com) as a subcontractor.

Integrating Plaggie into BOSS course management tool would be an interesting direction of further work. After all, BOSS was originally developed for programming courses. Anyway, we will probably make no major changes to Plaggie in the near future because it works well enough for the needs of our laboratory.

5. ACKNOWLEDGEMENTS

We thank Doctor S. Schaeffer for initiating the project and supervising the work in 2002.

6. REFERENCES

[1] Bowyer, K.W., and Hall, L.O. Experience using 'MOSS' to detect cheating on programming assignments. In: *29th ASEE/IEEE Frontiers of Education Conference*, San Juan, Puerto Rico, pp. 18-22. 1999.

[2] Grune, D. *The software and text similarity tester SIM.* Retrieved on October 9, 2006, from the Vrije Universiteit Amsterdam web site: http://www.cs.vu.nl/~dick/sim.html.

[3] Helsinki University of Technology. *Plaggie: Download*. Available at the Helsinki University of Technology web site: http://www.cs.hut.fi/Software/Plaggie/. 2006.

[4] Lancaster, T., and Culwin, F. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education, 14*, 2, pp. 101-117. 2004.

[5] McCauley, R., and Manaris, B. *Comprehensive Report on the 2001 Survey of Departments Offering CAC -accredited Degree Programs*. Technical report CoC/CS TR# 2002-9-1, Department of Computer Science, College of Charleston, 2002. Retrieved on February 11, 2004, from the College of Charleston web site: http://stono.cs.cofc.edu/~mccauley/survey/report2001/CompRep2001.pdf.

[6] Prechelt, L., Malpohl, M., and Phlippsen, M. JPlag: Finding plagiarism among a set of programs with JPlag. *Journal of Universal Computer Science, 8*, 11, pp. 1016-1038. 2002.

[7] SourgeForge.net. *Plagiarism Prevention Tools for Teachers.* Retrieved on October 14, 2006, from the SourgeForge.net web site: http://sourceforge.net/projects/turnitin.

[8] SourgeForge.net. *Sunlight: Source code similarity measure.* Retrieved on October 14, 2006, from the SourgeForge.net web site: http://sourceforge.net/projects/sunlight.

[9] SourgeForge.net. *The BOSS Online Submission System.* Retrieved on October 14, 2006, from the SourgeForge.net web site: http://sourceforge.net/projects/cobalt.

[10] University of Chile. *DetectaCopias 1.0.* Retrieved on August 25, 2006, from the University of Chile web site: http://www.dcc.uchile.cl/~rmeza/proyectos/detectaCopias/index.html. [In Spanish.]

[11] University of Sydney. *The Sherlock Plagiarism Detector*. Retrieved on October 14, 2006, from the University of Sydney web site: http://www.cs.usyd.edu.au/~scilect/sherlock/.

[12] University of Warwick. *History of BOSS.* Retrieved on October 14, 2006, from the University of Warwick web site: http://www.dcs.warwick.ac.uk/boss/history.html.

[13] Wagner, N.R. *Plagiarism by Student Programmers*. Retrieved on August 25, 2006, from the University of Texas at San Antonio web site: http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html. 2000.

Educational Pascal Compiler into MMIX Code

Evgeny A. Eremin
Perm State Pedagogical University
614990, Russia, Perm,
Sibirskaya, 24
eremin@pspu.ac.ru

ABSTRACT

The pedagogical software tool, visualizing translation from Pascal into machine codes, is described. The basic aim of the present work was to create software, distinctly demonstrating students the close unity of the contents of two fundamental CS courses – programming and computer architecture.

Keywords

Education, learning, Pascal, compiler, MMIX.

Intensive development of software tools increases the distance between a real executable program in processor instructions and its text on a high-level language. Specialists with long computer experience have constantly perfected their knowledge parallel to the development of calculating machinery. As they naturally progressed from processor codes to modern languages, they tried the full pallet of programming technologies. But now most people, who first begin studying high-level languages, can't imagine how programs they write are connected with processor instructions described in fundamental CS books.

The professional compilers have no purpose to generate demonstrative code and visualize it before a student. So special educational software [1] developed by the author is offered for these aims. It shows the accordance between any simple Pascal program and its executable equivalent in MMIX code (MMIX is a modern model of RISC computer, designed by Knuth [2,3]). A similar version for Intel processors' code also exists.

The suggested educational freeware for Windows media [1] can translate some subset of Pascal language which includes all basic algorithmical structures and run the result of compilation (virtual MMIX machine is realized inside the software to execute the output code). The compiler processes all standard Pascal data types (integer, real, char, boolean) and arrays of them.

The listed above possibilities allow to explain students the following high-level language features:

- variables, constants and type constants realization and the difference between them;
- memory organization in the form of array;
- converting relational Pascal types into each other;
- algoritmical structures on high and low level

and some other fundamental programming language basics.

Let us consider how to use the compiler for learning array indexing as an example. Figure 1 shows translation of the small demo program that works with arrays.

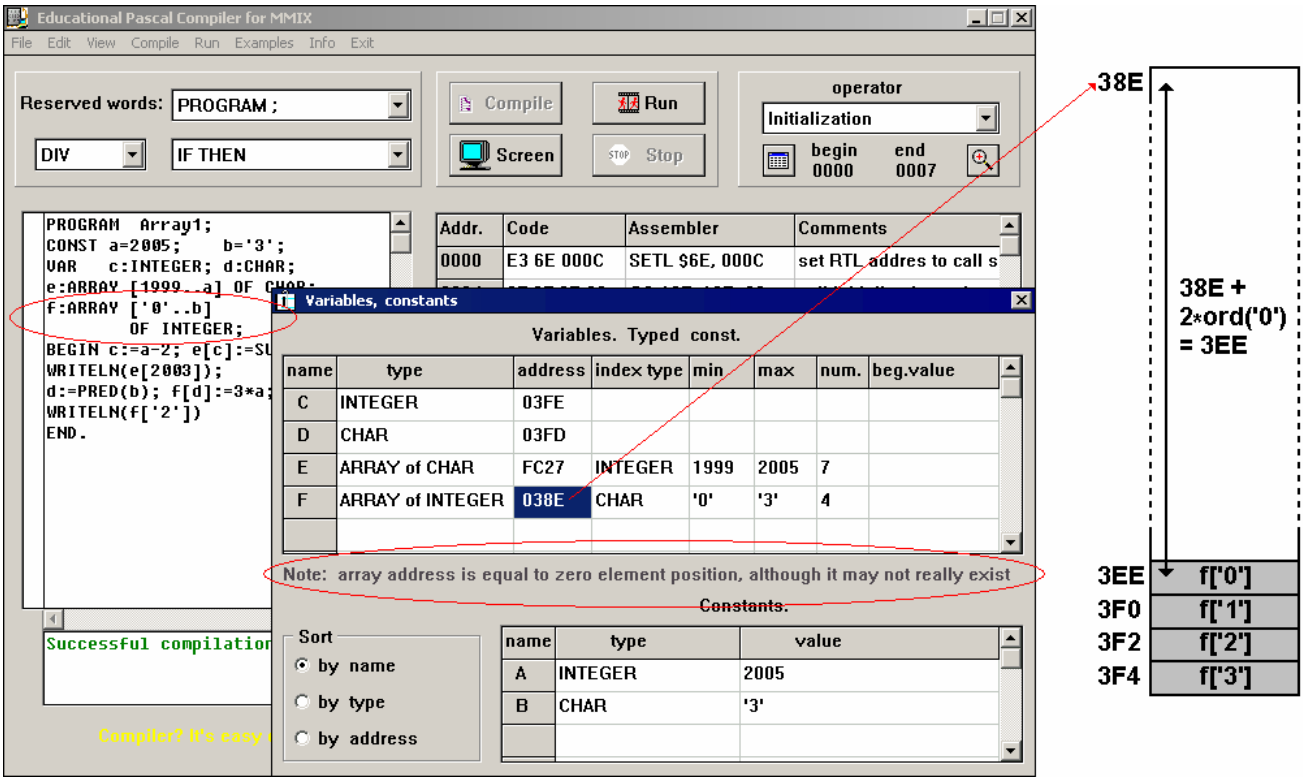

Figure 1. Using compiler to study arrays.

Note that this is only one of more than fifty samples built into the software; besides students can type their own programs.

The variables' description in this Pascal program announces that integer array named *f* has one index of char type, and its values belongs to '0'..'3' range. It is much more usable for further calculations to store the address of zero index although such element actually does not exist (in our case address is equal to 38E as it results from Figure 1). Then to calculate current address, for example *f[d]*, compiler can execute the following computation (see Table 1).

Table 1. Calculations of address for array

Address	Code	Assembler	Operation	Comment
104	E30303FD	SETL $3, 3FD	3FD → $3	*d* address
108	81030300	LDB $3, $3, 0	($3) → $3	load value
10C	39030301	SL $3, $3, 1	left shift $3	*2 - integer
110	E300038E	SETL $0, 38E	38E → $0	0-base
114	20000003	ADD $0, $0, $3	$0+$3→ $0	*f[d]* address

Analyzing this code, students can clearly see how array indexing is organized in the computer's memory.

The suggested educational compiler can also optionally generate standard fragment of program code that implements reasonability check of index *d* current value. This feature may come in useful for learning compiler fundamentals too.

To demonstrate how simple for understanding the compiled machine code of total Pascal program is, let us review the easiest example of the program here.

```
PROGRAM sample;
CONST x = 2;
VAR   y: INTEGER;
BEGIN y := x*x;
      WRITELN(y)
END.
```

The results of sample's compilation into MMIX code are presented in Table 2 (gray color in the table groups lines with isolated Pascal operators).

Table 2. Equivalent MMIX program with comments

Address	Code	Assembler	Operation	Comment
0	…			RTL library
B4	E30003FE	SETL $0, 3FE	3FE → $0	***y*** address
B8	E3010002	SETL $1, 2	2 → $1	constant ***x***
BC	19010102	MUL $1, $1, 2	$1*2→$1	***x*** * ***x***
C0	A5010000	STW $1, $0, 0	$1 → ($0)	store to ***y***
C4	E30103FE	SETL $01, 3FE	3FE → $1	***y*** address
C8	85710100	LDW $71, $1, 0	($1)→$71	load value
CC	9F6F7010	GO $6F, $70, 10	print integer	WRITE ***y***
D0	9F6F7034	GO $6F, $70, 34	next line	LN
D4	00000000	TRAP 0	exit	END

The contents of the table seem to be compact and clear. The conventional indirect scheme is used to exchange with any Pascal variable: first put its address into a processor register (marked by '$' in MMIX notation) and then make a reference through it.

We must additionally mention that a little Run-Time Library (RTL) is placed in the beginning of the code. The discussed sample needs RTL for printing the result of calculations to virtual MMIX display. Output RTL subroutines use registers $6F - $71 by definite simple arrangements. To make RTL code shorter and clearer, the realization of some standard algorithms (like converting float or integer numbers into string) are displaced into virtual ROM, so small pieces of code in the library just organize correct subroutine callings.

The software not only shows MMIX code in the above table, but gives some additional possibilities to students: they can extract any operator for examining (including search by run-time error address), analyze the map of variables, inspect system RTL and ROM.

A clear Windows interface allows even the beginners to work with this program. Dropdown list boxes with ready-to-use reserved words and initial standard template for the programs which a student can change make input process quick and comfortable.

One of the software versions has been used for several years at our University. In spite of some non-essential difficulties in the beginning, the results are positive: all students understand the basic compilation ideas. Most of them singly come to the conclusion that universal automatically generated code is not optimal and human translation is better. We discuss this problem and organize the competition for the shortest code with an advanced group of students. Another interesting form of task is to predict the results of compilation without computer and then check these predictions using the compiler.

The traditions of teaching in our country require a profound understanding of learning material. So the described software is meant to extend conventional high-level programming courses first of all. Working with educational compiler, students can acquire a wider outlook in computer languages foundations. Such pedagogical approach closely unites two fundamental CS disciplines – programming and computer architecture, which modern students often misapprehend as unrelated.

REFERENCES

[1] Educational Pascal Compiler into MMIX Code. http://www.winsite.com/bin/Info?27000000037259.

[2] Knuth, D.E. *A RISC Computer for the Third Millennium*. Heidelberg, Springer-Verlag, 1999.

[3] MMIX 2009 a RISC computer for the third millennium. http://www-cs-faculty.stanford.edu/~knuth/mmix.html.

Student Errors in Concurrent Programming Assignments

Jan Lönnberg
Helsinki University of Technology
P.O. Box 5400
Finland
jlonnber@cs.hut.fi

ABSTRACT

This poster abstract describes the ongoing work at Helsinki University of Technology on observing defects in concurrent programming assignment submissions and examining the underlying causes of these errors and the methods used to find them. The work is part of a larger endeavour to find problematic aspects of debugging concurrent programs and develop approaches to aid programmers in this task.

Keywords

Concurrent Programming, Computer Science Education, Bugs, Errors, Defect Cause Analysis

1. INTRODUCTION

Students' solutions to programming assignments provide material that can be used to improve several interlinked processes. The student-submitted assignment solutions (or *submissions*) can be used to evaluate and improve the students' learning, the teaching and the assignments. Information on defects in students' programs can also be used as a starting point for the development of debugging methodology and tools. Concurrency further complicates the programming process by introducing nondeterminism and its effect on debugging has seen little research.

For the reasons outlined above, I am examining submissions from the three programming assignments on the concurrent programming course at Helsinki University of Technology. The first involves writing control code for simulated trains that communicate using semaphores. In the second and third, the student implements and applies the Reactor pattern [5] and tuple spaces (respectively). The first data was collected during the Autumn 2005 course[1], and more detailed data will be collected during the Autumn 2006 course[2]. In 2005, students were required to submit both the actual program source code and a brief report outlining how their solution works with an emphasis on concurrency-related behaviour. Defects were found in the programs using a combination of testing and manual analysis and students' explanations of how their code works were used to deduce the underlying mistakes.

The work described here can be considered to belong to two different areas of research: research on defects in programs (e.g. [4]) and research on student errors in computer science assignments (e.g. [3, 6]). The former work aims to improve the quality of software by understanding why programmers err ("What errors do programmers make? Why do they make them? How can we get rid of them?"), while the latter aims at improving the quality of teaching ("What are the deficiencies in the students' knowledge and skill and in the teaching? How can we detect and eliminate them?").

[1] `http://www.cs.hut.fi/Studies/T-106.420/main.html`
[2] `http://www.cs.hut.fi/Studies/T-106.5600/english.shtml`

2. APPLICATIONS

As noted above, information on the types of defects in students' programs can be applied to developing teaching, the assignments and the assessment thereof, and to the development of debugging tools and methodology.

2.1 Teaching and Assignments

The results of an assignment can be used to determine whether students are effectively learning what they should. In particular, if a large number of students has problems understanding and/or applying some relevant knowledge, the teaching of this knowledge needs to be improved.

If students, on the other hand, produce many defects unrelated to the subject matter they are being taught, the assignment may be testing the wrong knowledge and skills. If the defects can be traced to misconceptions about the assignment or the artificial environment in which it is done (if it exists), the students may be distracted from learning relevant matters by difficulties specific to the assignment. Penalising students for defects that are arguably caused by a badly-designed assignment rather than any problem the student may have is hardly just. Therefore, it is important to recognise or eliminate these defects.

An experienced grader can quickly spot common defects in the assignments he grades, as he knows what to look for. Information on common defects can therefore be very useful to new graders on a course as a substitute for actual experience (both general and assignment-specific). Information on the errors underlying a defect can be used to guess the error made even in the absence of explanatory reports or comments.

Automatic assessment of programming assignments is typically done by executing test runs on the code to be assessed and assigning a grade based on the number of tests that passed [1]. One of the problems with automated assessment is that it is hard to design tests that detect all common errors and distinguish between different types of error without empirical data from real students.

2.2 Testing and Debugging

Concurrency makes debugging harder, as concurrent processes often interact in unexpected ways that can be hard to trace (e.g. race conditions). Only a few debuggers (e.g. RetroVue [2]) are specifically designed to aid in debugging concurrent programs, and they do not seem to be

widespread. Having quantitative data on concurrent programming errors provides a background against which debugging methods and tools can be developed that address common real-world problems related to concurrency. This information is hard to get from commercial development.

3. METHODOLOGY

In this research, the actual debugging and some of the defect detection is done using a set of automated tests and traditional debugging tools (especially print commands) as a complement to reading the code and trying to understand how it works and looking for common mistakes. This is part of the assessment process for the submissions, as knowledge of the defects in the program is required to give a grade that accurately reflects the proficiency of the student in concurrent programming and to provide constructive feedback on the defects (if any) in the submitted program.

The assignments have been designed in such a way that the solutions will be similar in structure. This means that many of the defects in the submissions can be considered to be the same defect in the algorithm that the program implements, providing a natural way of grouping many of the defects. The defects are further grouped by the part or aspect of the program affected and what the underlying error (mistake or misconception) was.

The errors made by students on the 2005 course were determined by examining the explanations provided by the student(s) in the form of code comments and the report explaining their reasoning. This information was collected to determine whether the defects were caused by slips in the implementation phase, design errors or misconceptions about concurrency or programming in general. In practice, only a few causes could be *confirmed* based on the students' explanations; in many cases, due to insufficient explanations, only a probable cause or probable causes could be determined.

4. PRELIMINARY RESULTS

The following preliminary results are based on the data from the Autumn 2005 course. Roughly half of the defects found (40 %, 60 % and 36 % in the respective assignments) appear to be cases of students misinterpreting what they are supposed to achieve. In the first assignment, only 15 % of these apparent misinterpretations could be confirmed based on student's explanations in their reports and comments (the others may be e.g. slips), but 44 % of them were confirmed in the second and third assignments. Based on the observed defects and explanations, most goal misunderstandings involve writing solutions that work correctly but use inter-thread communication methods other than those allowed in the assignment. In the second assignment, misconceptions about the Reactor pattern [5] are also clearly a problem. The latter can probably be mitigated by providing clearer material on the Reactor pattern. The former can be strongly discouraged by changing the environment in which the assignments are done to reflect the requirements instead of having seemingly arbitrary limitations on what the student may do (e.g. by setting up a real distributed tuple space in assignment 3 instead of requiring the student to write code that runs in a single process but communicates only through the tuple space). Such modifications have been made to the 2006 assignments; the defect statistics will show whether the modifications are successful. Decreasing the amount of misunderstandings of the goals of the assignments is useful in many ways: it allows information relevant to debugging of concurrent programs to be gathered more effectively (less irrelevant defects) and less of students' and teaching assistants' time is wasted on problems irrelevant to the learning goals of the course.

The simple train simulator used in the first assignment is directly related to 41 % of the defects in that assignment; 89 % of these are off-by-one errors in sensor positioning with many possible underlying causes. Only a few (5 %) of these could be confirmed to be misunderstandings of simulator behaviour or slips. This assignment also had the lowest proportion (12 %) of defects clearly related to concurrency (resource allocation between trains and other incorrect assumptions about interactions between trains). The other two assignments had 29 % (52 % confirmed) and 27 % (22 % confirmed) respectively: mostly failures to take into account all possible orderings of operations in different processes or limit them using synchronisation constructs.

The number of defects for which an exact error could be confirmed was quite low (23 %, 45 % and 34 % for the respective assignments); this suggests that asking students to explain the reasoning behind their entire solution in a written report does not give enough information to reconstruct their errors. In order to improve this in the 2006 data, students submitting corrected code after failing the assignment will be required to explain the reasoning behind the defective code. In order to get more information about their debugging methodology and the difficulties they face, they must also provide information about the measures they took (tools and debugging approach used, time taken) to track down and correct the defects.

5. ACKNOWLEDGEMENTS

This work was supported by the Academy of Finland (under grant number 210947) and Tekniikan edistämissäätiö.

6. REFERENCES

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] J. Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[3] L. Grandell, M. Peltomäki, and T. Salakoski. High school programming — a beyond-syntax analysis of novice programmers' difficulties. In *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, pages 17–24, 2005.

[4] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[5] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[6] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student errors in algorithm simulation exercises. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 81–86. University of Joensuu, November 2005.

Spatial Data Algorithm Extension To TRAKLA2 Environment

Jussi Nikander
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland
jtn@cs.hut.fi

ABSTRACT

In this paper an extension that brings spatial data algorithm support to the TRAKLA2 system is described. The different exercise types and the problems implementing them are discussed, and suggestions for future work and ideas for better spatial data support to the system are given.

Keywords

Spatial Data Algorithms, Geometric Algorithms, Teaching, TRAKLA2, Automatic Assessment

1. INTRODUCTION

TRAKLA2 [4] is an automated assessment system for teaching data structures and algorithms. The TRAKLA2 exercises are *visual algorithm simulation* exercises, where a group of data structures are illustrated to the user using *algorithm visualization* techniques. In visual algorithm simulation the user manipulates graphical representations with a mouse, thus simulating the actions of an algorithm. Typically the graphics are used to represent data structures. Most exercises in the TRAKLA2 system are *tracing exercises* where the user traces the execution of a given algorithm by manipulating graphical representations of data structures [3]. A typical modification in a tracing exercise is to move a data element from one data structure to another, or from one position in a given structure to another position in the same structure.

TRAKLA2 has so far covered only basic data structures and algorithms. This work describes a current project that aims to extend the idea of visual algorithm simulation to another area of algorithmics. This first extension to TRAKLA2 covers *spatial data algorithms* (SDA) in the field of geoinformatics. Geoinformatics is a branch of science that applies computer science techniques to cartography and other geosciences. Spatial data is a term that covers all data that is located in a multidimensional space. Spatial data algorithms are, therefore, algorithms that manipulate this situated data.

The SDA extension to the TRAKLA2 system covers the basic spatial data algorithms taught to geoinformatics students at Helsinki University of Technology, and is designed to be used on the SDA course arranged by the Institute of Cartography and Geoinformatics at the university. Many of the algorithms taught at the course may be better know to computer scientists as geometric algorithms.

2. SPATIAL DATA ALGORITHMS

In geoinformatics, spatial data algorithms are used to manipulate located, geographic data. Typically the data forms some type of map. Map data can be represented using two models: vector and raster. In the vector model, the different terrain features are represented as points, lines, or polygons that have location data associated to them. In the raster model, a regular grid is used to divide the represented area into equal sized cells. Each cell then has a data value associated to it. Spatial data algorithms can be divided into two categories according to which data model they use.

In addition to the two data models and the associated algorithms, there are also several data structures designed to store spatial data. Many such structures can, at least with minor modifications, be used to store data of either model. These structures are typically based on data structures designed to store one-dimensional data such as numbers or strings. For example, an R-Tree [2] is two-dimensional variation of the B-Tree.

2.1 Algorithms for Vector Model

In the vector model, the data elements are discrete, and in many cases cover only part of the whole space being examined. Such data can be gathered, for example, by taking samples in the field, and measuring the required data values at these sampling points. Such data is typically modeled using mathematical models, like Voronoi-diagrams [6]. Since the data covers only small part of the area, interpolation and extrapolation methods are important for achieving required coverage.

It is easy to create tracing exercises for most vector model algorithms. For example, in the expanding wave method for creating a TIN-model (a dual of the Voronoi-diagram) [5], the placement of the input points decides the order in which the data points are covered and therefore the execution of the algorithm. In such an exercise, the learner must, after a data point has been covered, decide which point to handle next based on the given input data.

Some of the data structures and algorithms used for handling spatial data in the vector model also demonstrate well-known problem solving methods that can be applied in other fields. The set of implemented exercises includes, for example, finding line segment intersections using the well-known line sweep problem solving strategy.

A total of 12 TRAKLA2 exercises that contain data structures and algorithms for the vector model have been specified for the SDA course. These exercises cover areas such as different structures for storing Voronoi-diagrams, methods for constructing, modifying and analyzing Voronois, methods for traversing and analyzing polygon maps, interpolation methods, etc..

2.2 Algorithms for the Raster Model

In the raster model, the whole area being examined is covered by a raster. Such data can be gathered, for example, by digitizing existing maps or from aerial photographs. The data is stored as a matrix of values.

Most of the algorithms used for the raster model include a lot of mathematical calculations. Typical operations are, for example, combining information from two raster layers to a new layer, or generalizing information in one layer. Such operations can be effectively implemented by using filter functions that calculate the values for a new raster layer based on existing values.

In most TRAKLA2 exercises the user traces the execution of an algorithm by moving data elements from one position into another. In most raster problems, however, the algorithm reads the input and uses a mathematical function to calculate the output. Such operations can not be simulated using tracing, but would require other kind of exercises.

Implementing most raster model problems in TRAKLA2 system would require considerable extra work to modify and expand the user interface. These expansions to the system have currently been left as future work. Therefore, only one exercise specific to the raster model has been specified for TRAKLA2.

2.3 Visualization of Multidimensional Data

Typically, in TRAKLA2, the data structures are visualized in a way that explicitly shows the internal hierarchy of the structure. A tree, for example, is visualized using a view where nodes are arranged to layers based on their distance from the root node, and each node has links to its child nodes. Root is on the topmost level. Such visualization shows accurately how the data is stored in the data structure. However, if the data is multidimensional, such visualization cannot accurately show the relationships (i.e. distance, direction, size) between different data elements in the given space.

In order to show the relationships between different data elements, a view that shows the given space is required. Typically a two-dimensional space is visualized as an area, where the different data elements are drawn to their correct coordinates. However, such view does not contain information necessary for understanding how the data is stored in the structure. Therefore, in the spatial data algorithm exercises, there is typically a need to show two views of the same data structure simultaneously: one to show how the data is stored and another to show how data elements are related.

3. CONCLUSIONS AND FUTURE WORK

Tracing exercises are meaningful when the input of the algorithm affects its control flow. In such exercises the input data must be used to decide what the next step of the algorithm is, forcing the learner to think how to execute the next step of the algorithm simulation. Therefore, in order to solve the exercise correctly, he must know how the algorithm works, and be able to solve the different special cases he encounters. Conversely, if the input has not effect on the control flow of the algorithm, a tracing exercise should not be used. When the steps of the algorithm are the same on each input, the solution process becomes a systematic task, where the learner does not need to think about the next step once he knows how the algorithm works. This, in turn, makes the exercise boring and repetitive.

Knowledge of many vector algorithms can be tested using tracing exercises. However, most raster algorithms are best suited for some other type of exercise. Knowledge of raster algorithms can be tested, for example, by having the user construct a desired filter function. Such exercises are traditionally done either with pen and paper, or by using mathematical programs such as Matlab. However, trying to expand visual algorithm simulation to cover such *exploratory* exercises [3] is an interesting research problem. One possible solution is to borrow ideas of visual programming environments such as Alice [1], and create exercises where the goal is to graphically construct a correct filter function using a given input and a set of possible operations.

4. REFERENCES

[1] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bulletin*, 35(1):191–195, 2003.

[2] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[3] A. Korhonen and L. Malmi. Taxonomy of visual algorithm simulation exercises. In *Proceedings of the Third Program Visualization Workshop*, pages 118–125, The University of Warwick, UK, July 2004.

[4] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267 – 288, 2004.

[5] M. J. McCullagh and C. G. Ross. Delaunay triangulation of a random data set for isarithmic mapping. *The Cartographic Journal*, 17(2), December 1980.

[6] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, 2000.

Creative Students – What can we Learn from Them for Teaching Computer Science?

Ralf Romeike
University of Potsdam
Department of Computer Science
A.-Bebel-Str. 89
14482 Potsdam, Germany
romeike@cs.uni-potsdam.de

ABSTRACT
Supporting creativity is a learning objective in general school education. In computer science the creation of products plays an important role. We want to find out the impact of creative action in computer science education classes on task motivation and understanding of computer science concepts. Underlying elements and environments of the creative process are analyzed by interviewing creative students. First results indicate creativity as an important factor for motivation. Furthermore the use of a concept we call "building blocks" helps to achieve creative products.

1. INTRODUCTION
Creativity is a phenomenon of human behavior, which is generally valued, admired and desired from politics and industry. With that the encouragement of creativity found its way into schools and curricula. It is also aimed for as a superior learning objective in computer science education.

Boden [1] describes two aspects of creative achievements. Historical creativity (H-creativity) describes ideas, which are novel and original in the sense that nobody has had them before. Something that is fundamentally novel to the individual Boden describes as psychologically creative (P-creativity). In an educational context the latter is more interesting. Thus the difference between an exceptionally creative person and a less creative person is not a special ability. It is based on a larger knowledge in a practical and applied form as well as on the will to acquire and use that knowledge.

Empirical studies show that today's professionals criticize the lack of creativity and other social skills in their education. Industry also remarks a lack of creativity in school graduates. School now is trying to fulfill the needs and educate the students more creatively.

We believe that the school subject of computer science can contribute a lot to that issue. Even if the transfer of creativity within different domains is discussed controversially [8], domain specific creativity still should be promoted. Computer science has the potential to keep up with subjects like music, arts or languages, which traditionally are said to enhance creativity. Perhaps it can even perform better in that.

2. CREATIVITY IN CS EDUCATION
Computer science, as computer scientists see it, is a creative field to work in [5]. There is also seen potential for creativity in computer science education. Especially modeling in the context of software development is emphasized as being a creative process [6,9]. It offers students "the possibility to develop creative solutions for problems, to realize and to validate them" [9] and should be revealed as a creative process to the students. It is questionable if this potential is actually applied in models for teaching and in the lessons. Approaches and justifications for computer science education generally tend to focus more on the aspect of problem solving or modeling of a problem with having the implementation of the solution as a motivating bonus at the end.

A lack of awareness for creative student activity is also found in lesson drafts which are presented in journals for computer science education. In an analysis of possible creative student action in 15 lesson drafts only two were explicitly assigning time for that. Sometimes it is even feared that students let their creativity out to much [4]. Obviously noncreative students are easier to handle in class than creative students. This can lead to making creativity even undesired. We think that explicitly planning of creative lesson phases is important. That means instead of only focusing on the recognition of principles of information systems there also needs to be time for designing and shaping them. Divergent learning tasks need to be offered to the learners. The goal of implementation should not be just a testing of paper-found or presented solutions and models, but needs to be implemented actively into the learning process. From lesson observations we assume, that such creative phases can enhance students motivation and raise understanding of computer science concepts. According to Scragg [7], students also have to learn creativity, which comes from insight into the facts, methods, and paradigms of the field. We intend to conduct research on the question if there is interdependency; if creative action conversely will have positive effects to the understanding of content, concepts and their interaction.

3. APPROACH
In our study we ask the question: Does the integration of creative phases in computer science classes enhance students' motivation and which effect do they have on their understanding?

In order to conduct creative lessons and empirically test the outcome we need to find out what creative work in computer science classes is about, especially what it means to students. We need to find out answers to the question: What are the underlying elements and environments of creative working in computer science classes?

Therefore we want to investigate experiences and attitudes of students, who stand out due to creative achievements in the lessons. How is the creative action connected with motivation and interest? Which understanding do these students have of creativity? Which role does creativity play for them? How do they perceive computer science? Which methods are they using?

First hints of possible answers about how to achieve creative products can be found in common literature of psychology and software development. We suggest asking the students in an outline based interview generally about their methods, approaches, strategies and views to computer science. If necessary, methods suggested in literature can be proposed and asked to be evaluated.

4. A PRE-STUDY

In a pre-study for exploring our field of research an outline interview was conducted with a student whose creative achievements stood out in and outside of the lessons. His actions we call creative due to the independent finding and formulation of new ideas (fluency), the change of categories for the subject he worked with (flexibility) and the development of personal novel products, without having a fixed solution at hand before (creative problem solving). During the interview programming came out as the major source of interest in computer science and thus was the main topic in the interview.

Practical work, in particular programming, was considered to be more interesting to the student. In working practically he enjoys the feeling of being able to be creative. With little effort he can accomplish a lot. The main source of motivation comes out of an aiming for self-realization. This is perceived as creativity and results in a final product.

For the student programming is a creative task. He states that it is creative, when a personal method is used for producing something. This understanding of creativity relates to the definitions of P-creativity and problem solving. To problem solving it adds the aspect that there is an idea about how a result may look like, but it is basically just a direction and not concretely defined. Thus the discovery of the direction, different possibilities and the uncertainty about the result play an interest raising role. In comparison with other school subjects, he argues that computer science is as creative as music and arts, even if this is not generally perceived the same by the majority of other students.

Constructs for programming are perceived as “building blocks”. Utilizing these building blocks programming appears to be easy. In this sense programming means putting blocks together as putting together bricks when building a house. In this case the bricks are fundamental programming constructs, such as variables, commands, iteration and so on. The principles and concepts of programming are applied when putting these bricks together.

The construction kit concept is well known in computer science: Data types are modelled using the building block principle; for the modelling of processes the different programming paradigms with their construction kits are used. Another construction kit would be e.g. the catalogue of fundamental ideas, which describe important concepts and strategies.

Similar results were found by Bruce et al [2] when they observed that students use building blocks in a programming course as an approach to gain understanding of the concepts as well as an approach for writing programs.

In contrast to the structured approach to programming in many didactic concepts the student often claims not to follow a certain strategy. More important is the possibility to try out things and proceed intuitively. The compiler permanently guarantees feedback which influences the problem solving process positively. The space of possible solutions in this way can be tested and the knowledge about it is extended. Glass makes a similar point about the use of “selective trial and error” in software development [3].

5. CONCLUSIONS

The majority of concepts and theories about how to teach computer science in school are based on deductive approaches. Starting from the computer science perspective, especially software development, promising content and concepts are carried over and adapted to the classroom. In contrast our approach starts bottom-up. Applying the experiences of creative students a catalogue of hints for creative working shall be developed. Promising first results suggest motivation and understanding are closely related to creative action and vice versa. In teaching concepts this aspect does not find enough attention. In continuing research on that issue we try to provide a basis to allow students to harness their creativity.

To receive a more reliable collection of elements supporting creative action more interviews need to be done. Therefore the requirements for selecting creative students need to be concretized, a variety of students need to be chosen and the interview outline designed.

Finally the findings need to be applied in an empirical study to draw conclusions about the effect of creative phases in the lessons.

6. REFERENCES

[1] Boden, M. A. *The creative mind: myths & mechanisms*. Basic Books, London, 1990.

[2] Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M. and Stoodly, I. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, 3:143-160, 2004.

[3] Glass, R. L. *Software Creativity*. Prentice Hall PTR, Englewood Cliffs, 1995.

[4] Janneck, M. *Partizipative Systementwicklung im Informatikunterricht.* LOG IN 138/139, LOG-IN-Verlag, Berlin, 2006, 60-66.

[5] Leach, R. J. and Caprice A. A. The Psychology of Invention in Computer Science. In *Proceedings of the Psychology of Programming Interest Group Workshop.* Brighton, UK, 2005.

[6] Schulte, C. *Lehr-Lernprozesse im Informatik-Anfangsunterricht : theoriegeleitete Entwicklung und Evaluation eines Unterrichtskonzepts zur Objektorientierung in der Sekundarstufe II*. Diss. Univ. Paderborn, 2003.

[7] Scragg, G., Baldwin, D. and Koomen, H. Computer science needs an insight-based curriculum. *ACM SIGCSE Bull*., 26(1):150-154, 1994.

[8] Sternberg, R. J. *Creativity: From potential to realization*. APA, Washington, DC, 2004.

[9] Thomas, M. *Informatische Modellbildung: Modellieren von Modellen als zentrales Element der Informatik für den allgemeinbildenden Schulunterricht*. Diss. Univ. Potsdam, 2002.

Recent technical reports from the Department of Information Technology

2007-005 Parosh Aziz Abdulla, Noomene Ben Henda, Richard Mayr and Sven Sandberg: *Stochastic Games with Lossy Channels*

2007-004 Jonas Persson: *Pricing American Options Using a Space-time Adaptive Finite Difference Method*

2007-003 Pierre Flener, Justin Pearson, Magnus Ågren, Carlos Garcia Avello and Mete Çeliktin: *Air-Traffic Complexity Resolution in Multi-Sector Planning*

2007-002 Jing Gong and Jan Nordström: *A Stable and Efficient Hybrid Scheme for Viscous Problems in Complex Geometries*

2006-052 Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno and Ahmed Rezine: *Regular Model Checking without Transducers (On Efficient Verification of Parameterized Systems)*

2006-051 Erik Bängtsson and Björn Lund: *A Comparison Between Two Solution Techniques to Solve the Equations of Linear Isostasy*

2006-050 Iordanis Kavathatzopoulos: *AvI-enkäten: Ett verktyg för att mäta användbarhet, stress och nytta av IT-stöd*

2006-049 Stefan Blomkvist: *The User as a Personality: A Reflection on the Theoretical and Practical Use of Personas in HCI Design*

2006-048 Owe Axelsson, Radim Blaheta and Maya Neytcheva: *Preconditioning of Boundary Value Problems using Elementwise Schur Complements*

2006-047 Henrik Johansson and Johan Steensland: *A Performance Characterization of Load Balancing Algorithms for Parallel SAMR Applications*

2006-046 Mei Hong, Torsten Söderström and Wei Xing Zheng: *Asymptotic Accuracy Analysis of Bias-Eliminating Least Squares Estimates for Identification of Errors in Variables Systems*

2006-045 Anders Hessel and Paul Pettersson: *Model-Based Testing of a WAP Gateway: an Industrial Case-Study*

2006-044 Mei Hong, Torsten Söderström, Johan Schoukens and Rik Pintelon: *Comparison of Time Domain Maximum Likelihood Method and Sample Maximum Likelihood Method in Errors-in-Variables Identification*

2006-043 Jarmo Rantakokko: *Case-Centered Learning of Scientific Computing*

2006-042 Eddie Wadbro: *On the Far-Field Properties of an Acoustic Horn*

2006-041 Markus Nordén: *Performance Modelling for Parallel PDE Solvers on NUMA-Systems*

2006-040 Mei Hong, Torsten Söderström and Wei Xing Zheng: *A Simplified Form of the Bias-Eliminating Least Squares Method for Errors-In-Variables Identification*

2006-039 Andreas Hellander and Per Lötstedt: *Hybrid Method for the Chemical Master Equation*

2006-038 Markus Nordén, Henrik Löf, Jarmo Rantakokko and Sverker Holmgren: *Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers*

UPPSALA
UNIVERSITET

February 2007
ISSN 1404-3203
http://www.it.uu.se/