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 *Research* papers present unpublished, original research, presenting novel results, methods, tools, or interpretations that contribute to solid, theoretically anchored research.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.

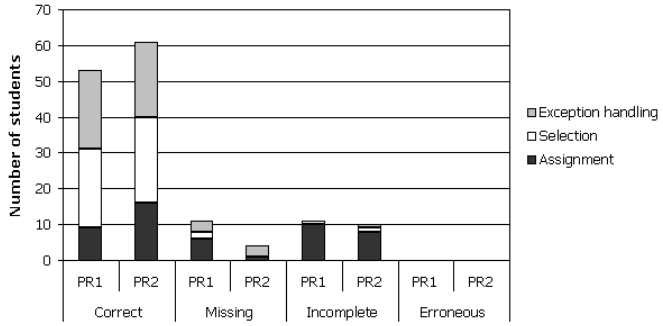

Figure 1: *The frequency of different types of explanations given by students for individual statements in the two progress reports.*

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".

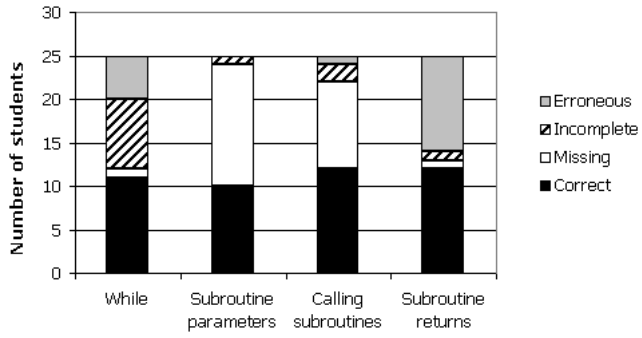

Figure 2: *The frequency of different types of explanations given by students for new topics in the second progress report.*

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.

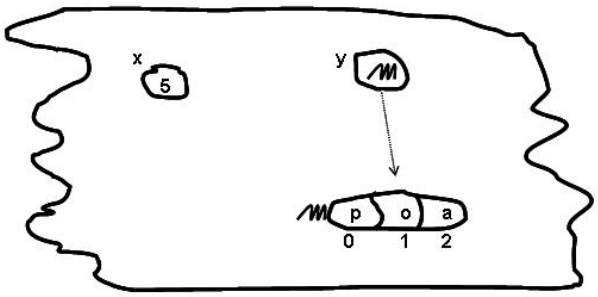

Figure 1

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.

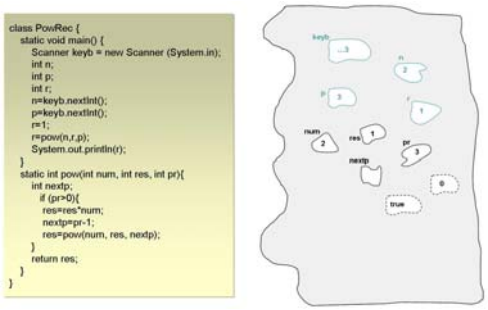

Figure 2

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.