Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2004 TKO-A42/04

Kolin Kolistelut — Koli Calling 2004

Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education

October 1–3, 2004 in Koli, Finland
Organized by the University of Joensuu, Finland

Ari Korhonen and Lauri Malmi (editors)

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Distribution:
Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
P.O. Box 5400, 02015 TKK
FINLAND

Tel: +358 9 451 4809
Fax: +358 9 451 3293
Email: sihteeri@cs.hut.fi
URL: http://www.cs.hut.fi/

ISBN 951–22–7438–8
ISSN 1239–6885

Otamedia Oy
Espoo 2004

Foreword

This is the proceedings of the Fourth Finnish / Baltic Sea Conference of Computer Science Education. In Finland, the conference has already acquired a well-established position as a meeting point for CS educators and researchers of CS education. However, the conference is gaining more international visibility, and this year we had 37 participants from 7 countries.

The primary goal of the conference is to promote the development of CS education and CS education research. The surrounding Koli nature park with its wonderful sceneries and beautiful nature has its special role in providing us a relaxed forum for sharing thoughts and new ideas.

The program included two invited lectures. First, Anders Berglund from the Uppsala University gave a tutorial on different research approaches in computer science education research. Second, professor Patrik Scheinin from the University of Helsinki gave a keynote address "Evaluating the effect of teaching". Both of these lectures gave us inspiring thoughts how we could improve our research work in the field. Especially we should increase our knowledge on research methodology that can be applied in CS education research. With such knowledge we are better equipped to evaluate whether the new educational methods and tools really improve the learning results of our students.

This year the program committee decided to call separately for research papers and discussion papers to make a clearer distinction between papers that present novel ideas, approaches and systems for CS education, and papers in which these issues have been elaborated further in some rigid research setting. Both types of papers are, however, equally necessary for the whole CS education community. New ideas and tools are the fuel for research work, and research is needed to convince us that we are really making progress towards our goal of improving learning.

We received 30 submissions including 16 research papers, 13 discussion papers and one demonstration paper. All research papers were reviewed by three members of the program committee, and all discussion papers and posters by two PC members. I wish to present here my thanks to all of them, since they managed to complete the review process within the very tight two weeks schedule.

To improve the objectivity of the review process, each paper was reviewed by at least one PC member outside Finland, and no one reviewed papers that were submitted from his/her own institute. We finally decided to accept 10 submissions as research papers, 12 submissions as discussion papers, and 5 submissions as posters / demonstrations.

The Koli tradition has been that the discussion in the conference is an important part to improve the papers even further. Thus, the authors should gather the feedback, ideas, and suggestions they receive in the conference, and incorporate them into the final version of the paper, which appears in the proceedings. For this reason, the program committee also decided that all papers should be presented in the conference. Otherwise they were not accepted in the proceedings.

I hope that you will enjoy reading this proceeding, and that the papers will inspire new thoughts and ideas about improving CS education, which is our common goal.

Espoo, Finland, November 2004

Lauri Malmi

Program committee

Lauri Malmi, Helsinki University of Technology, Finland (chair)
Anders Berglund, Uppsala University, Sweden
Michael E. Caspersen, University of Aarhus, Denmark
Valentina Dagiene, Vilnius University, Lithuania
Jaakko Kurhila, University of Helsinki, Finland
Guido Rößling, Darmstadt University of Technology, Germany
Tapio Salakoski, University of Turku, Finland
Erkki Sutinen, University of Joensuu, Finland

Organising committee

Ilkka Jormanainen, University of Joensuu, Finland (head)
Jaana Lukkarinen, University of Joensuu, Finland (local arrangements)

Editors

Ari Korhonen, Helsinki University of Technology, Finland
Lauri Malmi, Helsinki University of Technology, Finland

Contents

Keynote Addresses

Research paper presentations

Discussion paper presentations

Poster presentations

PART 1
Keynote addresses

Teaching and learning CS - An overview of research approaches

Anders Berglund
Department of Information Technology
Uppsala University
SWEDEN anders.Berglund@it.uu.se

Abstract

The results that can be obtained from a research project on students' learning of computer science are closely related to how the research is performed. The relation is complex: Similar results, or, at least, results illuminating the same research question, can be obtained in different ways. Neither does the selection of a particular way of performing the research necessarily lead to a certain type of result. The researcher himself or herself - his or her beliefs, interests, previous experience, network of discussants, even his or her personality - are essential for the outcome of a project and become, to a certain degree, part of the results. Other factors that influence the outcome are the setting in which the study is performed, previous work within the domain and previous work performed in a similar way, to name just a few. A *research approach* is essential to guide the researcher as he or she studies students' learning of computer science. It serves as a "tool box", and offers established guidelines for conducting empirical work, as well as a theoretically sound perspective on learning. An approach can be compared to a lens in that some aspects are in focus, while others become unclear or "blurred". The selection of a research approach is crucial, since a particular approach offers certain perspectives on the research question, and enables in this way the researcher to study certain aspects of learning. With a conscious selection of a research approach, communication with other researchers becomes possible as a shared terminology is available. In this way, the researcher obtains the instruments with which to judge the relevance of his work in the light of research performed by others, and, of course, to judge the findings of others in relation to his or her own results. Furthermore the approach helps in understanding to what extent the results can be trusted and generalized to other groups of students and to other situations. The deployment of an approach in a particular project, that is, a selection of which methods that are used (for example, to collect data) are not defined by an approach, but are instead methodological decisions that are taken by the researcher based on the current situation, of which the approach and the research questions are key elements. Of course, since a research approach has a history, including other projects - more or less successful - performed within that approach, there is a tradition, or a network of competence, that the researcher partly can lean on in his or her selection to use one method (for example interviews for data collection) over another (for example observation). In this tutorial, I will discuss some research approaches that are relevant for research into students' learning of computer science. Examples of results that can be obtained from the different approaches are given.

Evaluating the Effects of Teaching

Patrik Scheinin
Department of Educational Sciences
University of Helsinki
FINLAND

patrik.scheinin@helsinki.fi

1 Introduction

The greatest strength and at the same time weakness of education as a science lies in the fact that everybody is an expert. We all have a vast amount of experiences of parenting, teaching at school, and of studies. This makes research results interesting and understandable to a broad audience. Problems arise, however, when individual experiences and anecdotal evidence is generalized. An educated guess is mostly fine when we make decisions about our own education. Its limits and flaws become more obvious when we plan education for others and encounter the variation in ability, attitude, and knowledge of our students. An opinion based on personal experience is likely to strengthen the beliefs of the likeminded, but it is of little use in convincing the opposition. This is where research using efficient and powerful designs plays a role. Good research can be measured by how convincing it is to critical opponents. And if it is worth doing at all, it is worth doing well.

2 Research designs

2.1 Surveys

The term survey typically refers to questionnaires, interviews, or a combination of these. Survey methodology measures variables by asking people questions and then examining response levels or relationships among the variables. In most instances, surveys measure beliefs and attitudes. The most common type of survey uses the cross-sectional design, in which the data is gathered at one point in time.

Surveys can be a cost-effective type of research. All surveys are basically exploratory. If the sample is representative of the target population and the questions asked are to the point and easy to understand, the results may give a valid picture of how positive or negative the measured beliefs or attitudes are more generally. Surveys are much more fallible when the researcher uses the correlations between the measured variables to analyze causal relationships. For understanding cause-and effect and ruling out rival hypotheses, experimental or quasi-experimental research is needed.

2.2 Experiments and quasi-experiments

The basic difference between experimental or quasi-experimental research is that experimental designs use random assignment to groups or programs, whereas the quasi-experimental designs do not. Quasi-experiments use control groups or multiple measures of the same group, and may use various methods to try to insure the similarity of the studied groups or to control for differences between them. Quasi-experiments are often the best available solution in educational research, as many experiments are either not possible to carry out for practical reasons, or ethically questionable. We may for instance be highly interested in the development of study skills and attitudes of students in different subjects, but it would not be realistic to delegate students randomly to their studies.

In the following examples X represents any form of intervention or effect that is to be studied. These could be variations in teaching method, the instructional material used, or

the way the students were motivated. O represents observation. Observations could include quantitative measures of skills, knowledge or attitudes, but observations of behaviour in class or interviews might also be used.

Our first example is actually not an experiment at all. It does, however, unfortunately represent a common design brought on by the fact that educators typically first develop and apply their method and only at a much later stage begin to think about evaluating the results of their improvements. At this stage it is hard to say anything about changes brought on by the intervention. Students might be asked if they feel they have changed during the intervention, but obviously their recollections may be coloured by many factors.

Example 1

$$X \quad O$$

A variation of the previous design is to gather data both before and after the studied intervention. Many problems remain. Most importantly, we do not know if the measured change between pre-test O1 and post-test O2 would have occurred even without the intervention e.g. due to age typical development during university studies. A second possibility for confusion has to do with the group becoming test wise after having practiced with the pre-test. Attitudes may be influenced if those who have been tested begin to pay more attention to certain aspects of their lives. Often comparison between pre and post results is further complicated by attrition. If, for example, students drop out from the course we study, the results (before and after) may reflect the fact that only the most motivated or clever students remained. In this case the mean of the mean of the pre-test results could be significantly lower than those of the post-test, even without any real positive intervention effect.

Example 2

$$O_1 \quad X \quad O_2$$

Our third design remedies some of the aforementioned problems. Its main strength lies in having a control group. This will enable the researcher to compare normal development within the control group with the development and intervention effect within the experimental group. Differences between the groups could be partly controlled by pre-testing, as well as by trying to match the subjects of the groups. These remedies make comparisons more trust-worthy, but between-group differences may still be systematic and confound the intervention effects.

Example 3

$$\begin{array}{ccc} O_1 & X & O_2 \\ O_1 & & O_2 \end{array}$$

The best designs for evaluating effects of teaching involve random assignment of subjects to experimental group and control group. The R below indicates randomisation. Theoretically all possible independent variables are controlled. Thus, the pre-test is not needed for control purposes (4a). If we are mainly interested in finding out if our intervention has any effect design 4a is adequate. A pre-test gives us a chance to compare normal development with that induced by the studied intervention (4b). If we wanted to find out what effect our radically improved course has on the development of programming skills in our students design 4b would be called for.

Example 4a

$$\begin{array}{ccc} & X & O \\ R & & \\ & & O \end{array}$$

Example 4b

$$\begin{array}{cccc} & O_1 & X & O_2 \\ R & & & \\ & O_1 & & O_2 \end{array}$$

One further improvement might be suggested. When three groups with random assignment are used, we see the difference between post-test results of the experimental and the control groups. We can also see how much our subjects have developed during the intervention. In addition to this, the inclusion of the third group lets us control for the possible effects of participating in the pre-test on the post-test results.

Example 5

$$\begin{array}{cccc} & O_1 & X & O_2 \\ R & O_1 & & O_2 \\ & & & O_2 \end{array}$$

Even the most sophisticated designs are powerless in the face of certain human weaknesses. When by a certain person carries out the intervention, the effect of the intervention is likely to be confused with that of the said person (often an exceptionally competent and inspired teacher). A study indicating the effectiveness of a certain teaching method may be invaluable all over the world, but a study indicating the efficiency of some colleague, who cant be transferred, is mostly of limited value. Adding parallel experimental and control groups to the design and assigning them to different teachers is one way of handling the problem.

It is especially important to note that even design 5 would be seriously flawed if various forms of selective attrition occurred in the different groups. Analysis of attendance rates and the selectiveness of attrition should always be included in experimental studies. Finally, almost any change, any extra attention, any manipulation of external circumstances, and most especially changes in instruction, or just the knowledge that a study is being done, can be enough to cause non-intended changes in the outcome. If students know that a nice teacher is evaluating a certain intervention into which she has invested a lot of effort, the students are likely to try to help their teacher by improving the results.

2.3 Longitudinal designs

Longitudinal designs ask the same questions at two or more points in time. These designs look at how beliefs and attitudes change over time. There are two main types of longitudinal design. The time-series or trend study is basically a repeated cross-sectional design. A time-series design collects data on measures representing a population on the same variable at regular intervals. Time-series designs are useful for establishing a baseline measure and its variation, describing changes over time, keeping track of trends, and forecasting the short term future. Panel designs collect repeated measurements from the same people or subjects over time. Panel data can show different patterns from time-series data. For example, only a minority of the university students has problems with their studies at any given time, but it is not always the same people. So students have a much larger risk of encountering problems at some point of their studies.

You make observations to establish a baseline, do an intervention, and then make some more measurements. The number of observations you make before or after the intervention is determined by the stability of the data.

Example 6

$$O_1 \quad O_2 \quad O_3 \quad O_4 \quad A \quad O_5 \quad O_6 \quad O_7$$

The problem with design 6 is that other events may be confounded with the intervention effect. Typically, development tends to occur in the subjects in educational settings. A second parallel series may be added as a comparison group. Ideally, the experimental treatment would be randomly assigned to one group or the other. The addition of the comparison group allows the researcher to rule out extraneous factors as explanations for the observed results.

Example 7

$$\begin{array}{ccccccccc} O_1 & O_2 & O_3 & O_4 & & O_5 & O_6 & O_7 \\ O_1 & O_2 & O_3 & O_4 & A & O_5 & O_6 & O_7 \end{array}$$

2.4 Reversal or ABAB design

When control of extraneous variables is an especially important concern, and multiple groups cannot be managed, the best alternative may be a reversal or alternating-treatments design. Baseline data is optional but recommended if circumstances allow. The essence of this design is a random presentation of at least two forms of treatment, e.g. normal instruction versus some new model, and continuous monitoring of the effects. Usually the different treatments will be provided from session to session. The order for the different treatments may be randomized to increase the likelihood that the observed effects are the result of the treatment rather than some extraneous influence.

Example 8

$$O_1 \quad A \quad O_2 \quad B \quad O_3 \quad A \quad O_4 \quad B \quad O_5$$

3 Conclusions

The aim of this paper has been to provide a frame of reference for research design of present and future studies in Computer Science Education. We may conclude that the designs used in the majority of the studies are less than optimal for their purpose. The reasons are obvious. Mostly it is a question of saving money or other resources. Typically in computer science the focus has been on program development and the evaluation research was added on as an afterthought. A weak design does, however, often not provide the answers looked for in the study, or the answers may not be generalizable. In such cases one may ask if the saving were altogether too expensive. Randomised designs with control groups will hopefully be common enough to be standard practice in the field in the near future. Even the most elegant research design will, however, fail miserably if selective attrition occurs. While a random attrition of more than 60 attrition of far less than 10 study. Analysis of attendance rates and the selectiveness of attrition should, therefore, always be included in experimental studies.

Bibliography

Advances in measurement in educational research and assessment. (1999) Edited by Geoffrey N. Masters and John P. Keeves. Amsterdam : Pergamon.

Educational Research, Methodology, and Measurement: An International Handbook. (1988) Edited by J.P. Keeves. Oxford: Pergamon.

Foundations of behavioral research, 4. ed. (2000) Edited by Fred N. Kerlinger and Howard B. Lee. South Melbourne : Wadsworth.

Robson, C. (2002) Real World Research, 2. ed. Oxford: Blackwell.

PART 2
Research paper presentations

Questions, Annotations, and Institutions: observations from a study of novice programmers

Robert McCartney
Dept. of Computer Science and Engineering, Univ. of Connecticut, Storrs, CT, USA

Jan Erik Moström
Department of Computing Science, Umeå University, Umeå, Sweden

Kate Sanders
Dept. of Math and Computer Science, Rhode Island College, Providence, RI, USA

Otto Seppälä
Dept. of Computer Science and Engineering, Helsinki Univ. of Technology, TKK, Finland

robert@cse.uconn.edu

Abstract

This paper examines results from a multiple-choice test given to novice programmers at twelve institutions, with specific focus on annotations made by students on their tests. We found that the question type affected both student performance and student annotations. Classifying student answers by question type, annotation type (tracing, elimination, other, or none), and institution, we found that tracing was most effective for one type of question and elimination for the other, but overall, any annotation was better than none.

1 Introduction

In summer, 2004, a working group at the ITiCSE conference in Leeds, UK, examined the code reading and understanding of novice computer programmers. The analysis was based on data collected from twelve institutions in Australia, Denmark, England, Finland, New Zealand, Sweden, the United States, and Wales. These data were based on a multiple-choice test administered to beginning programming students, and included the student answers for all of the questions. For a subset of the students, interview transcripts and the actual test forms with any annotations used during the test were also collected. The working group analyzed a broad range of issues: the kinds of questions, the performance of students by institution and quartile, the sorts of annotations used and their general effectiveness, student test-taking strategies observed, and others(Lister et al., 2004).

In this paper, we look in detail at a small slice of these issues: the interrelationships observed between the kinds of annotations used by the students, the style and difficulty of the individual questions, and the institutions where the students were tested. In particular, we would like to determine how the likelihood of answering a question correctly is affected by the various kinds of annotations used.

2 Classifying annotations

An annotation, referred to as a "doodle" by the Working Group, was defined to be any kind of marking by a student on his or her exam paper. In Figure 1 we can see an example where the user made a number of marks: numbers written over variables, numbers written under array elements, and many assignment statements setting variables to constants. In this case, the assignments show the student keeping track of variables as the loop is executed–a graphical record of the tracing process.

Two of the Working Group members did a data-driven classification of the doodles. This classification was later independently verified by three other members (see (Lister et al., 2004) for details). The classification is given in Table 1. Using this, the researchers classified 56 complete exams: the three from each institution corresponding to the students who were

interviewed about their tests, plus 20 others chosen at random from the six researchers who had other tests with them in Leeds.

Table 1: Categorization of annotations. % is percentage of questions showing each type.

Name	Code	Description	%
Blank page	B	No annotations for this question	38
Synchronized trace	S	Shows values of multiple variables changing, generally in a table	11
Trace	T	Shows values of a variable as it changes (more than 1 value) or a variable's value is overwritten with new value	32
Odd Trace	O	Something that appears to be a trace but neither S or T, such as linking representations with arrows	3
Alternate answer	A	Student changed their answer to the question	4
Ruled out	X	One or more alternative answers crossed out so it answer appeared to be selected by elimination	9
Computation	C	An arithmetic or boolean computation (not rewrite of comparison)	4
Keeping tally	K	Some value counted multiple times, variable not identified	1
Number	N	Shows single variable value, most often in comparison	28
Position	P	Picture of correspondence between array indices and values	11
Underlined	U	Part of question underlined for emphasis	7
Extraneous marks	E	Markings that appear meaningless or ambiguous (could not be characterized). Includes arrows, dots, and so forth	13

As the example in Figure 1 shows, an answer can contain several different doodle categories: N, P, and T for this question.

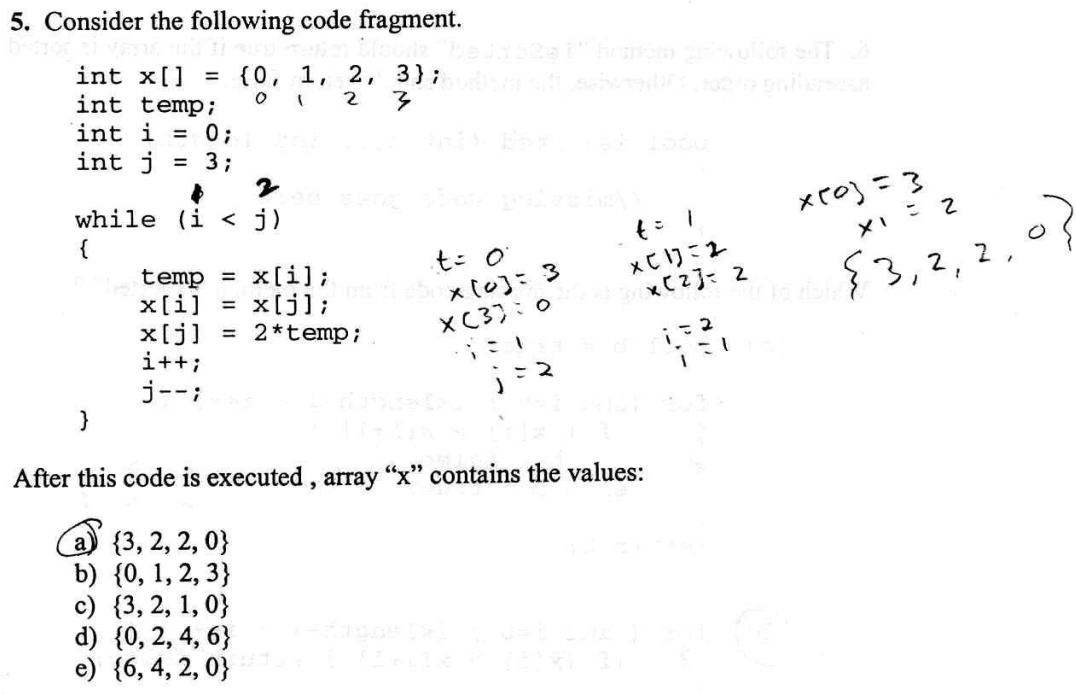


Figure 1: Example of doodles on test from question 5, showing annotations N, P, and T.

3 Classifying the exam questions

The multiple-choice exam used for this project consisted of twelve questions involving a variety of array-processing tasks, such as comparing two arrays in order to find elements that are contained in both, testing to determine whether an array is sorted, filtering some of the elements of one array into another, searching for a given element in an array, and deleting the element in a given position from an array.

Each question on the test can be classified into one of two categories, *fixed-code* questions, where the student is given a code fragment and asked questions about the result of executing it, and *skeleton-code* questions, where the student is given an incomplete code fragment, and asked to complete it so it will perform a given task. There were seven fixed-code and five skeleton-code questions on the exam.

Questions 3 and 5 in Figure 2 are representative fixed-code questions. They give a single piece of code and ask the student about what is true after the code is executed. The style of all the fixed-code questions is the same as these: "Consider the following code fragment: *[code fragment]* What is the value of *[some int or array variable]* after this code is executed?" And each of the possible answers is a constant, either the value of an integer variable (as in question 3) or a list of the values in an array (as in question 5). Other than code, they require very little reading.

Question 8, also shown in Figure 2, is a representative skeleton-code question. Like Question 8, the skeleton-code questions generally provide a description of what the code is to accomplish, a code fragment containing one or more blanks, and a set of choices to fill in the blanks. The choices are all code fragments themselves, as opposed to the numeric values used in the answers to the fixed-code questions.

The students' performance was noticeably different on the two types of questions. Overall, our students had an average score of 61%, indicating that they do not have a strong grasp of the basic knowledge in these areas. Students did better on the fixed-code questions than on the skeleton-code questions, however. On the fixed-code questions, from 61% to 73% of all the answers were correct (depending on the question). On the skeleton code questions, the percentages were 35%, 43%, 46%, 58%, and 72%–all but one of them worse than any of the fixed-code questions.

4 Analysis

Two issues complicated our analysis of these data. First, the large majority of the annotations were done on the fixed-code questions, as can be seen in Table 2. The difference between the fixed-code and skeleton-code questions is striking: 77% of the fixed-code questions are annotated, as opposed to 41% of the skeleton-code questions. Indeed, if we were to group the questions on the basis of how often they are annotated, without looking at the questions themselves, we would have the same groups: Questions 1-5, 7, and 10, the fixed-code questions (68%-88% annotated) and Questions 6, 8-9, and 11-12, the skeleton-code questions (38%-45% annotated). Because the two groups were annotated so differently, there was the danger

Table 2: Number and and percentage of annotated questions observed. These are based on analysis of 672 questions (56 of each question).

	Question											
	1	2	3	4	5	6	7	8	9	10	11	12
count	38	44	40	48	49	23	41	25	22	40	25	21
percent	68%	79%	71%	86%	88%	41%	73%	45%	39%	71%	45%	38%
Q type	FC	FC	FC	FC	FC	SC	FC	SC	SC	FC	SC	SC

that overall conclusions would be determined by the data from the annotations of the fixed-

Question 3. Consider the following code fragment:

```
int [] x = {1, 2, 3, 3, 3};
boolean b[] = new boolean [x.length];
for ( int i = 0; i < b.length; ++i )
    b[i] = false;
for ( int i = 0; i < x.length; ++i )
    b[ x[i] ] = true;

int count = 0;

for (int i = 0; i < b.length; ++i )
    if ( b[i] == true ) ++count;
```

After this code is executed , "count" contains:

a) 1
b) 2
c) 3
d) 4
e) 5

Question 5. Consider the following code fragment:

```
int [] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;
while (i < j) {
    temp = x[i];
    x[i] = x[j];
    x[j] = 2*temp;
    i++;
    j--;
    }
```

After this code is executed, array "x" contains the values:

a) {3, 2, 2, 0}
b) {0, 1, 2, 3}
c) {3, 2, 1, 0}
d) {0, 2, 4, 6}
e) {6, 4, 2, 0}

Question 8. If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array "x" that contains the following six numbers:

```
4   5   6   2   1   3
```

There are 10 inversions in that array, as:

```
x[0]=4   >   x[3]=2
x[0]=4   >   x[4]=1
x[0]=4   >   x[5]=3
x[1]=5   >   x[3]=2
x[1]=5   >   x[4]=1
x[1]=5   >   x[5]=3
x[2]=6   >   x[3]=2
x[2]=6   >   x[4]=1
x[2]=6   >   x[5]=3
x[3]=2   >   x[4]=1
```

The skeleton code below is intended to count the number of inversions in an array "x":

```
int inversionCount = 0;
for ( int i=0 ; i<x.length-1 ; i++ )
{
   for  ( xxxxxx )
   {
      if (  x[i] > x[j] ) ++inversionCount;
   }
}
```

When the above code finishes, the variable "inversionCount" is intended to contain the number of inversions in array "x". Therefore, the "xxxxxx" in the above code should be replaced by:

```
a)   for ( int j=0   ; j<x.length   ; j++ )
b)   for ( int j=0   ; j<x.length-1 ; j++ )
c)   for ( int j=i+1 ; j<x.length   ; j++ )
d)   for ( int j=i+1 ; j<x.length-1 ; j++ )
```

Figure 2: Questions 3, 5, and 8.

code questions. Accordingly, we considered the fixed-code and skeleton-code questions both together and as two separate groups.

The second issue that complicates this analysis is that the classifications are not disjoint; with the exception of Blank, any combination of annotations can occur on any given question. The observed non-blank questions had from 1 to 5 different annotation types represented, with an average of approximately 2. As an example of this problem, consider class N, numbering, which was relatively common (appearing in just over 28% of the questions). 90% of the time that there was numbering, however, there were other classes as well; 77% of the time at least one tracing type was also present.

To resolve this issue, we created four disjoint categories, reclassifying each question as

Blank, Some Tracing (S,T, and O, but not A or X), Elimination (A or X), or Other (everything else). The rationale for these categories is that tracing and process of elimination are recognizable strategies, and, with Blank, cover 89% of the observations.

We then counted the number and percentage of time the answers were correct for each category, for fixed-code questions, skeleton-code questions, and for all questions taken together. The results are given in the following tables.

Table 3: Percentages of annotation types for each question type. Numbers in parentheses present the counts (questions having this annotation, all questions).

Question type	Blank	Some tracing	Elimination	Other
fixed-code	23 (92 of 392)	61 (238 of 392)	6 (25 of 392)	9 (37 of 392)
skeleton-code	59 (164 of 280)	6 (17 of 280)	21 (59 of 280)	14 (40 of 280)
all	38 (256 of 672)	38 (255 of 672)	13 (84 of 672)	11 (77 of 672)

Table 4: Percentages correct, by question and annotation type. Numbers in parentheses present the counts (correct and total answers).

Type	Blank	Some tracing	Elimination	Other	Total
fixed	50 (46 of 92)	76 (180 of 238)	48 (12 of 25)	54 (20 of 37)	66 (258 of 392)
skeleton	50 (82 of 164)	65 (11 of 17)	61 (36 of 59)	55 (22 of 40)	54 (151 of 280)
all	50 (128 of 256)	75 (191 of 255)	57 (48 of 84)	55 (42 of 77)	61 (409 of 672)

These tables illustrate a number of things:

1. Overall, answers showing explicit tracing are the most likely to be correct: 75% are correct, compared with 50% for questions without annotation. We see similar results for FC and SC questions handled separately.

2. Overall, elimination is the second-most effective strategy.

3. Across the board, both overall and for the fixed-code and skeleton-code subsets of the data, any form of annotation, even Other, is better than no annotation at all.

4. The frequency of tracing is much lower for skeleton code questions than fixed code questions.Although tracing is used less, it is still highly effective for the skeleton-code questions when it is used.

5. The frequency of elimination is much higher for the skeleton code questions (where it is the most common annotation) than for the fixed code questions. Its effectiveness for skeleton code questions is slightly better than tracing, and much better than no annotations.

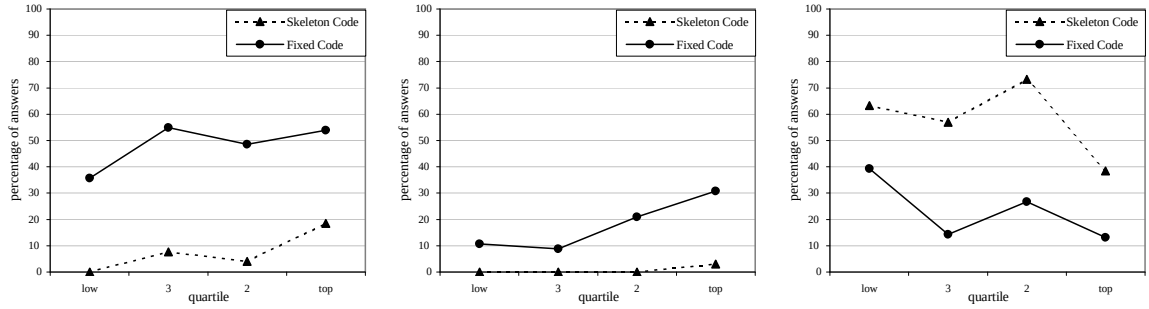
6. Skeleton-code questions are much more likely than fixed-code questions not to be annotated at all.

5 Annotations and overall performance

In the working group paper (Lister et al., 2004), the authors were able to extract differences between questions by observing what students in different quartiles (based on test score) chose in each question as their answer. We applied similar techniques here, with the expectation that students who do well on the exam would be more likely to use the effective annotation techniques. Results from three annotation types can be found in figure 3.

We found that tracing individual variables (T) was used about the same for the top three quartiles for fixed-code questions, but for skeleton-code questions, the top quartile was twice

as likely to trace as the second and third. The increased likelihood for the top quartile to use synchronized traces (S) relative to the lower quartiles was even greater, although the overall use was lower than tracing for the top quartile–possibly because not all questions required multiple variables to be traced. The data related to blank questions are less easily explained, however, as the frequency of blank questions does not increase monotonically as we go from the top to the bottom quartile – the second quartile students have relatively more blanks than the third quartile for fixed-code questions, and more than the third or fourth for skeleton-code questions.



(a) Tracing a single variable (b) Synchronized tracing (c) Blank

Figure 3: Percentage of an annotation type used in different quartiles

6 Annotations and institutions

While the popularity of annotations varied substantially from institution to institution, they led to higher scores in almost every institution. Table 5 shows the frequency of annotations for the 12 institutions (identified by letter due to confidentiality requirements).

Table 5: Percentage of questions with any annotations, by institution and question type.

Question type	Institution											
	E	J	L	S	Q	H	O	A	T	N	C	P
all	28	36	50	51	58	60	69	70	87	89	89	92
fixed code	36	55	62	71	71	89	81	89	98	86	100	100
skeleton code	17	10	33	23	40	20	53	43	73	93	73	80

These are large differences. Overall, the percentage of questions with annotations varies from 28 to 92. The percentages of fixed-code questions with annotations range from 36 to 100, and for skeleton-code questions, from 10 to 93.

To try to isolate the performance effects of annotation, we examined the performance difference between "strategic" annotations (tracing or elimination) and no annotations for each institution. (Tracing and eliminations were pooled since the numbers of observations at each institution are rather low). These data are given in Table 6. This table further supports the inference that students who annotate their exams tend to perform better: four of the five highest averages are from institutions in the top five in annotation frequency. In addition, in ten of the twelve institutions, annotated questions were more often correct than "blank" questions, which would indicate that the positive effect of tracing is fairly universal. (It should be noted that the number of blank questions is extremely low for institutions T,N,C, and P (9, 4, 4, and 3 respectively), so the comparisons for those institutions are of dubious value.)

Table 6: Percentage of questions correctly answered by institution, for Some tracing or elimination and blank. Average score is based on all questions at the institution.

	Institution											
	E	J	L	S	Q	H	O	A	T	N	C	P
Some tracing or elimination	87	77	87	54	72	59	63	76	78	63	76	58
Blank	62	48	33	66	53	21	45	38	67	100	50	0
Average score	68	57	58	60	61	42	56	63	78	67	75	39

7 Discussion: making sense of these results

Some of the results make obvious sense: for example, tracing through the code on paper helps; the proportion of correct answers where there is tracing is much higher than where there is not. One result seem counter intuitive at first glance: students use tracing less on the harder questions, where such strategies might be expected to be effective. We can offer three possible explanations:

Too much work Answering a skeleton-code question by tracing through the different alternatives would mean four or five times the work compared to a fixed-code question. Many students seem to realize this and instead chose to change their problem solving strategy: they begin by reading the question and forming a hypothesis of how the program should work, then look for boundary values that can be used to rule out different alternatives, and finally they check whether the selected alternative seem to work. This strategy would also explain the increase in eliminations for skeleton-code questions as shown in Table 3.

Too abstract The fixed-code questions are simple in the respect that they only require a knowledge of the syntax/semantics of the language and the ability to carefully trace the values of different variables. The skeleton-code questions are different in that they give a textual description of the problem, which the students have to translate into some problem understanding, they then have to form a hypothesis of a possible solution based on the alternatives, and then evaluate the different alternatives to find the correct one possible by creating one or more test cases (compare this to the software comprehension model described in (Boehm-Davis, 1988)).

Lack of representation Closely related to the explanation above is the lack of representation. As described above the student is required to *understand* the problem description for skeleton-code questions. This might be difficult to do without representing the task at the abstract level, a skill that most novices apparently do not have. It is observed in (Bransford et al., 1999) that novices tend to solve problems concretely (plugging values into equations, e.g.), while experts tend to apply the (correct) abstract principles–it may be that novices cannot represent the abstract version of the task.

From the available data we can not determine which, if any, of these are true, but answering this is an interesting possible research topic.

Regarding the different amounts of doodling at the different institutions, there are a number of possible causes–local culture, the way programming is taught, the way the test was administered, chance, and so forth. It may simply be that the subject pools were fairly homogeneous within institutions.

8 Related work

Davies (1993) found that novices and experts have different ways of annotating programs, and also that experts spend more time annotating than novices. Both Davies' and our results indicate that annotations are a successful strategy for finding the correct answer.

Thomas et al. (2004) found that students who drew object diagrams performed better on tests involving object references, but their attempts to encourage greater use of diagrams were largely ineffective. Indeed, even after the benefits of using diagrams were demonstrated to students, they failed to use them when taking exams.

Hegarty (2004) looks at relations between externally produced diagrams and internal visualizations. One of the relations that she proposes is the use of external visualizations to augment internal visualizations: some of the internal processing and memory is "off-loaded" to an external representation. Experimental evidence in (Hegarty and Steinhoff, 1997) showed that "low-spatial" subjects who made annotations on an external diagram performed as well as "high-spatial" subjects when doing problems involving the inference of mechanical component motion, which suggests that the use of external annotation can substitute for keeping track of details internally.

Perkins et al. (1989) identify tracing (which they refer to as "close tracking") as a fundamental skill required by programmers: even for novices, it can be used to avoid, diagnose, and repair bugs in programs. They also observed that students fail to trace their code effectively, and identify a number of reasons for such failure:

1. students do not understand that tracing can be useful, or are not confident that they can trace effectivly,
2. students do not understand the programming language primitives,
3. students "project intentions" onto the code, and reason from these rather than from the code itself, and
4. students have differing cognitive styles.

We explicitly observed the third reason, students "recognizing" what a code fragment does and reasoning from that level, and suspect that the other three also occurred in this study.

More generally, many previous relevant studies, specifically regarding novice programmers, use of strategies, and distinguishing comprehension and generation, are cited in a survey by Robins et al. (2003).

9 Conclusions

This project used both fixed-code and skeleton-code questions to test concepts and identify misconceptions in introductory programming students. Performance in both improves when students annotate their tests, particularly by tracing on paper. There are differences, however: fixed-code questions require only understanding of how each expression works. Skeleton-code questions are closer to writing code, and can require more abstract reasoning. As they require code to meet a specification, they require students to reason about aggregate function, determine proper test cases, and so forth. Other strategies, such as process-of-elimination, may be more appropriate here, as the space of four or five different pieces of code and a number of possible test cases may make tracing too tedious.

In addition to the questions as to why students annotate harder questions less, this work suggests some areas for further study:

Novice/expert annotation. Do the annotation patterns used by individuals change over time? Some previous work (Davies, 1993) suggests both the number and the type of annotations differs between novices and experts. Can similar differences be seen between first and last year students and what would these differences mean?

Institutional differences. Are institutional differences as large as these data (Table 5) would indicate? Do these differences reflect differences in how programming is being taught, or culture, or simply differences in the way the data were collected?

MCQs in practice. The results in the paper shows a clear difference between fixed-code and skeleton-code questions which makes them appropriate for different stages in a first programming course. How might we best exploit these differences to improve the student learning experience?

10 Acknowledgments

Thanks to all of the working group participants (see Lister et al. (2004)), and the local arrangements people at Leeds who provided a great work space and plenty of tea and coffee. Thanks to the organizers, reviewers and participants of *Kolin Kolistelut*: the organizers who provided an intellectually stimulating environment, and the reviewers and participants who asked good questions and offered good ideas on how to improve this paper and build on these results. Finally, thanks to Sally Fincher, Marian Petre, and Josh Tenenberg, whose Bootstrapping and Scaffolding workshops (supported by NSF grants DUE-0122560 and DUE-0243242) had a large positive influence on this work.

References

Boehm-Davis, D. A., 1988. Handbook of Human-Computer Interaction. Elsevier, Ch. 5 (Software Comprehension), pp. 107–121.

Bransford, J. D., Brown, A. L., Cocking, R. R. (Eds.), 1999. How people learn: Brain, mind, experience, and school. National Academy Press, Washington, D.C.

Davies, S., 1993. Externalising information during coding activities: effects of expertise, environment, and task. In: Empirical studies of programmers: 5th workshop. Ablex, Norwood, NJ, pp. 42–61.

Hegarty, M., 2004. Diagrams in the mind and in the world: relations between internal and external visualizations. In: Blackwell, A., Marriot, K., Shimojima, A. (Eds.), Diagrams 2004, LNAI 2980. Springer-Verlag, Berlin Heidelberg, pp. 1–13.

Hegarty, M., Steinhoff, K., 1997. Use of diagrams as external memory in a mechanical reasoning task. Learning and Individual Differences 9, 19–42.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J., Sanders, K., Seppälä, O., Simon, B., Thomas, L., 2004. A multi-national study of reading and tracing skills in novice programmers. SigCSE Bulletin (to appear).

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., Simmons, R., 1989. Conditions of learning in novice programmers. In: Soloway, E., Spohrer, J. C. (Eds.), Studying the Novice Programmer. Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 261–279.

Robins, A., Rountree, J., Rountree, N., 2003. Learning and teaching programming: a review and discussion. Computer Science Education 13 (2), 137–172.

Thomas, L., Ratcliffe, M., Thomasson, B., 2004. Scaffolding with object diagrams in first year programming classes: some unexpected results. In: Proceedings of the 35th SIGCSE technical symposium on computer science education. Norfolk, USA, pp. 250–254.

A multi-national, multi-institutional study of student-generated software designs

Sally Fincher[1], Marian Petre[2], Josh Tenenberg[3], Ken Blaha[4], Dennis Bouvier[5], Tzu-Yi Chen[6], Donald Chinn[7], Stephen Cooper[8], Anna Eckerdal[9], Hubert Johnson[10], Robert McCartney[11], Alvaro Monge[12], Jan Erik Moström[13], Kris Powers[14], Mark Ratcliffe[15], Anthony Robins[16], Dean Sanders[17], Leslie Schwartzman[18], Beth Simon[19], Carol Stoker[20], Allison Elliott Tew[21], Tammy VanDeGrift[22]

S.A.Fincher@kent.ac.uk

Abstract

This paper reports a multi-national, multi-institutional study to investigate Computer Science students' understanding of software design and software design criteria. Student participants were recruited from two groups: students early in their degree studies and students completing their Bachelor degrees. Computer Science educators were also recruited as a comparison group. The study, including over 300 participants from 21 institutions in 4 countries, aimed to understand characteristics of student-generated software designs, to investigate student recognition of requirement ambiguities, and to elicit students' valuation of key design activities. The results indicate that with experience, students become more aware of ambiguous problem specifications and are able to address more of the requirements in their software designs, that they use fewer textual design notations and more graphical and standardized notations, that they systemically ignore groupings and interactions among the different parts of their designs, and that students change their valuation of key design activities in response to changes in problem-solving context.

1 Introduction

Software design is difficult: dealing with ill-defined and ill-structured problems; having complex and often conflicting constraints; producing large, complex, dynamic, intangible artefacts; and being deeply embedded in a domain, such as finance or medicine (cf. characteristics of the design task described by Goel and Pirolli (1992)). As a result, software design requires a variety of skills and knowledge: within the domain of application, in programming (Soloway and Ehrlich, 1984), and in the mapping between the domain-based problem and software artefacts that carry out the requisite functionality (McCracken, 2004). This paper describes

[1]Computing Laboratory, University of Kent, UK
[2]Department of Mathematics and Computing, Open University, UK
[3]Computing and Software Systems, Institute of Technology, University of Washington, Tacoma, USA
[4]Department of Computer Science, Pacific Lutheran University, USA
[5]Department of Computer Science, Saint Louis University, USA
[6]Department of Mathematics and Computer Science, Pomona College, USA
[7]Computing and Software Systems, Institute of Technology, University of Washington, Tacoma, USA
[8]Department of Mathematics and Computer Science, Saint Joseph's University, USA
[9]Uppsala University, Sweden
[10]Computer Science Department, Montclair State University, USA
[11]Computer Science and Engineering, University of Connecticut, USA
[12]Computer Engineering and Computer Science, California State University Long Beach, USA
[13]Department of Computing Science, UmeåUniversity, Sweden
[14]Department of Computer Science, Tufts University, USA
[15]Department of Computer Science, University of Wales Aberystwyth, UK
[16]Computer Science Department, University of Otago, New Zealand
[17]Department of Computer Science/Information Systems, Northwest Missouri State University, USA
[18]School of Computer Science and Telecommunication, Roosevelt University, USA
[19]Mathematics and Computer Science Department, University of San Diego, USA
[20]Department of Computer Science, Azusa Pacific University, USA
[21]College of Computing, Georgia Institute of Technology, USA
[22]Computer Science and Engineering Department, University of Washington, Seattle, USA

results from a study of the software designs of over 300 Computer Science (CS) students and educators on a simple design task. (A fuller description of these results can be found in (Fincher et al., 2004).)

This study is distinctive from other studies of software design along a number of dimensions. First, it is both multi-institutional and multi-national, with participants from 21 institutions in 4 countries, one of the few software design studies with such a diverse participant pool. Only a multi-institutional study like this allows the assessment of what factors vary across educational contexts - and hence are likely to be influenced by educational intervention - and what are invariant. Second, the data that is examined is particularly rich, with the main components being the written representations and verbal descriptions of participant-generated software designs. This allows many diverse research questions to be addressed using multiple methods of analysis. Third, the study is large-scale, with over 300 participants, which, when combined with the study's multi-institutional nature reduces sample bias and increases generalizability. Given the cost and challenges of carrying out empirical research at this scale, there are few precedents for empirical software studies of this size and scope (but see (McCracken et al., 2001) and (Petre et al., 2003) for other such examples). And fourth, the study includes participants at three different levels of educational attainment, thus allowing the examination of changes in design behaviour with additional formal education.

2 Background

Looking at what Adams et al. (2003) call the *design expertise continuum*, we can gain insight into the different developmental stages of software designers which, it is to be hoped, can be incorporated into more effective design teaching and learning. Jeffries et al. (1981) noted that novices differ from experts in their ability to decompose a software problem effectively, to solve sub-problems, and to integrate solutions. Experts organise information differently from novices, producing different and larger "chunks" (summarised in (Kaplan et al., 1986).) In a study of industrial design engineers, Christiaans and Dorst (1992) found that novices tend to scope out a problem less and seek less information than experienced designers. Rowland (1992) found that novices made few requests for clarifications relative to a design problem.

Expert software practitioners have codified design expertise associated with robust, maintainable, testable, and flexible designs, often focusing on the interaction between different computational modules, as in the design principles of Bruegge and Dutoit (2000): "Ideal subsystem decomposition should minimize coupling and maximize coherence." But even when such principles are taught, it is far from clear that student designers have sufficient skill to apply these principles in practice.

In addition to studying expert/novice differences, some design researchers examine differences in student designers at different stages in their education. For example, Atman et al. (1999) studied differences in the design processes between freshmen and senior engineering students while developing designs for a playground. Not only were there differences in design quality between the two sub-populations, there were differences in design behavior as well. For example, seniors made more requests for information, made more than three times as many assumptions, and made more transitions between design steps, as compared to freshmen. Atman et al. (2003) also examined the design processes of engineering educators so as to provide insight into both educators' actual design practices and its implications for student learning.

In examining student conceptions of design, Newstetter and McCracken surveyed freshmen engineering students by having them rank the five most important and five least important from a list of 16 design activities. They found that the freshmen ranked as least important those activities that are central to general design process descriptions, (e.g. (Goel and Pirolli, 1992)) such as *decomposing*, *generating alternatives*, and *making trade-offs*. Adams et al. (2003), additionally provide evidence that expertise is characterized by matching the design

process to the design context: "experts do not approach every problem in the same way but rather adapt to the inherent constraints of the task."

3 The Study

This study used two tasks to explore students' understanding of the software design process: *a decomposition task*, to examine students' ability to analyse a problem and then design an appropriate solution structure, and to elicit students' understanding-in-action of fundamental software design concepts; and *a design criteria prioritization task*, to elicit which criteria students consider most and least important for different design scenarios.

3.1 Decomposition Task

Participants were given a one-page specification for a "super alarm clock" to help students manage their sleep patterns, and were directed to produce a design meeting these specifications. Participants were asked to "(1) produce an initial solution that someone (not necessarily you) could work from (2) divide your solution into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does - in short, why it is a part. If it is important to your design, you may indicate an order to the parts, or add some additional detail as to how the parts fit together." Participants performed this task individually, without communicating with peers or tutors. On completion, participants were asked to "talk through" their design, and to name and describe the function of each part.

3.2 Design Criteria Prioritization Task

After completing the decomposition task, participants were given 16 cards, each describing a single design criterion. (The phrases represented: Encapsulation, Implementability, High Cohesion, Loose Coupling, Chunking, Intelligibility, Explainability, Parsimony, Re-usability, Recognition of structure, Clarity, Design-phase testing, Maintainability, Engineering, Input re-use, Clear functionality. The phrases as presented can be found in (Fincher et al., 2004)).

Participants were asked to indicate the five most important and the five least important criteria for each of four scenarios. The first scenario was to with respect to the design they had just completed. The participants were then asked to rank the criteria (five most/least important) for each of three hypothetical scenarios:

- for the current task, but in a team (task in team),
- for the current task – on their own – but delivering a fully-functional result at the same time tomorrow (extreme time pressure), and
- for the current task, but designing the system as the basis of a product line that would have a 5-year lifespan (longevity).

3.3 Participants

Participants recruited from 21 institutions of post-secondary education from the USA, UK, Sweden and New Zealand completed the same tasks. Three types of participant were represented from each institution:

First competency students. (FC) To ensure comparability across institutions, students were selected at the point in their education where they could be expected to program at least one problem from the set proposed by McCracken et al. (2001). These problems involve the simulation of a simple calculator for arithmetic expressions. The McCracken problem set was used because it references levels of competence, irrespective of curriculum and was devised for use in one of the first multi-national, multi-institutional

CS Education Research studies. Not all of the FC participants were Computer Science majors, but all had taken, or were taking, a Computer Science course.

Graduating students. (GS) Graduating students were defined to be those within the last eighth of a Bachelor degree program in Computer Science or a related software intensive degree.

Educators. (E) Educators were defined to be those holding faculty positions, and teaching in the undergraduate program.

The total cohort consisted of 314 participants from 21 institutions representing 28 educators, 136 first-competency and 150 graduating students.

For each participant the following material was collected: their representation of the design, the time they took to make it, and a record of their prioritization of the design criteria. Full transcriptions of verbalisation during the task were made for a proportion of the students; researcher notes were made for all.

4 Results and Discussion

Three independent analyses were undertaken to provide different perspectives on the data that was collected. Each analysis is distinguished by the questions explored and the methods used. Exploratory, data-driven analysis of the design artefacts was undertaken to answer questions about the types and characteristics of representations that participants used. A directed qualitative analysis focussed on participants' recognition of ambiguity in the problem specification and in their information-seeking behaviour. And a quantitative analysis was used to answer questions concerning participants' prioritization of the design criteria.

4.1 Characterisation of Design Artefacts

4.1.1 Design Representations

This part of the study was a data-driven examination of the "marks on paper" representations. A sample of designs were first examined in order to develop a set of distinct categories into which each design representation would be grouped. As software practicioner-researchers, we developed these categories to represent semantically meaningful differences in design notation. The categories are:

Standard Graphical: recognised notations of software design, such as Class Diagram, or Entity-Relationship Diagram.
Ad-hoc Graphical: diagrams of any form that were not recognised as standard notations of software design.
Code or pseudo-code: code segments such as assignments, iteration and selection.
Textual: free text descriptions with at most an occasional illustrative diagram.

Each design artefact was visually examined, and categorised into exactly one of the previous disjoint groupings based on its predominating characteristic, or into the category **Mixed** if there was no clear dominance among the other categories.

To ensure consistency the designs were all categorised by three of the authors and assignment to a category required consensus. Figure 1 shows the results of this analysis. The data show a shift from textual to standard graphical representations with increases in education, with the frequency differences between the different subpopulations statistically significant at the $\alpha = .001$ significance level using the χ^2 test. While 47% of FC participants used predominantly textual representations, only 27% of GS participants and 23% of E participants did so. These numbers are inverted for standard graphical representations, with 46% of E participants, 27% of GS participants, and 15% of FC participants using these representations.

4.1.2 Design Complexity

Two indicators of design complexity were examined: the use of grouping structures among parts, and whether the design contained an indication of interaction among the parts. Each researcher analysed the designs from their own institution in terms of grouping by answering the question "Did the design include any hierarchical, nested, or grouping structure of any kind?" For example, a diagram with boxes labelled *Pocket PC*, *Alarm Handler*, and *User Interface*, collectively labelled as *User/Front End* would count as grouping. Similarly indication of interaction was analysed by each researcher answering the question "Are interactions between any of the parts indicated?" For example, a diagram with two boxes, an arrow linking the two boxes, and an explanation that one box is providing information to the other box would count as interaction.

There was some difference in frequency of use of grouping structures between the participant subpopulations; 24% of FC, 27% of GS and 46% of E participants used grouping, with the difference between the combined student groups and the educators significant at the $\alpha = .025$ significance level using the χ^2 test.

There was also significant difference in frequency of use of interaction between the participant subpopulations; 66% of FC, 81% of GS, and 93% of E participants indicated interaction, with the difference between these significant at the $\alpha = .001$ level using the χ^2 test, and significant at the $\alpha = .05$ level when the student groups are combined.

For both complexity measures there are marked institutional differences. For grouping it ranges from a low of 5% of the participants from one institution who included grouping in their designs to a high of 86% of the participants of another institution Q who did so. For those who indicated interaction in their designs, it ranges from a low of 40% of the participants from one institution to a high of 100% of the participants of three institutions who did so. This suggests that there is a strong effect on design depending on how this material is taught.

4.2 Recognising Ambiguity in Requirements

An analysis was conducted to investigate participants' recognition of ambiguous aspects of the design brief requirements. Recognizing and addressing ambiguity is important because it is cheaper to recognize and resolve ambiguities early, rather than after the design is completed (Boehm, 1981).

A participant is called an ambiguity *recognizer* if they ask a question or make an assumption, either in the written representation or verbally during the decomposition task. A participant is an *information gatherers* if they ask questions, whether or not they make observable assumptions. 216 participants are recognizers and 87 are non-recognizers, with 11

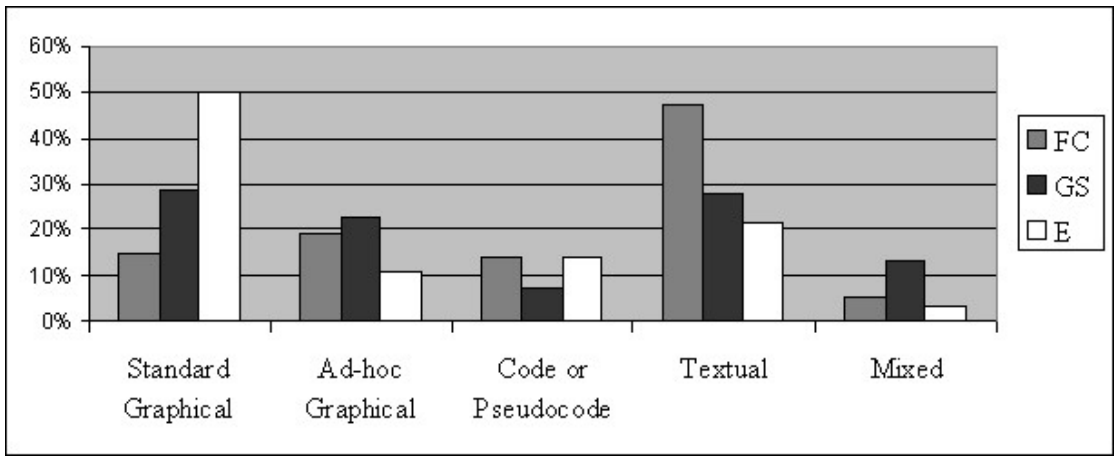


Figure 1: Distribution of Design Representations

participants not being reliably classified. The percentage of recognizers increases with education, with 63% of first competency students being recognizers, 76% of graduating seniors being recognizers, and 89% of educators being recognizers. There are 138 information gatherers and 165 non information gatherers, with no data for 11 participants. As with the recognizers the percentage of information gatherers increases with education; 33% of first competency students, 50% of graduating seniors, and 81% of educators gathered information during the decomposition task.

The information gatherers and recognizers were also analyzed with respect to the number of requirements addressed, with each participant classed into one of four categories: *all* requirements, *most* requirements (>= 50%), *some* requirements (< 50%) *no* requirements. The general trend is that as the number of requirements addressed decreases, the percentage of recognizers also decreases. This indicates that those who recognized ambiguity had a higher success rate in addressing all requirements than those who did not recognize ambiguity. Participants who gathered information also had a higher success rate in addressing all requirements than those who did not gather information. This trend indicates an association between recognizing ambiguity and successfully fulfilling requirements. Once again, institutional differences were evident. In five institutions all participants were recognizers, while in three institutions less than half of the participants were recognizers.

4.3 Design Criteria Prioritization

One of this study's focal questions is whether students recognize different criteria within the design process. This was motivated by discussion with educators and papers such as (CMM Correspondence Group, 1997), as well as textbooks such as (Bruegge and Dutoit, 2000) that suggest that there are particular criteria that should be considered when doing software design. The particular focus here is on the relative importance of different criteria and how these rankings vary over different participant groups and contexts. By examining these prioritizations across participant groups, it could be possible to see how (or whether) these are learned through the curriculum. These rankings were collected immediately after, and are set in the context of, the decomposition task. The data for each prioritization were collected into frequency counts for each participant group and scenario.

Figure 2 shows the number of times each criterion was ranked as one of the five most important criteria by each participant group for the design task that the participants just completed. There is a surprising similarity of valuations across participant groups. Similar such graphs for both "most" and "least" prioritizations in each of the four scenarios can be found in (Fincher et al., 2004); for reasons of space they are not included here.

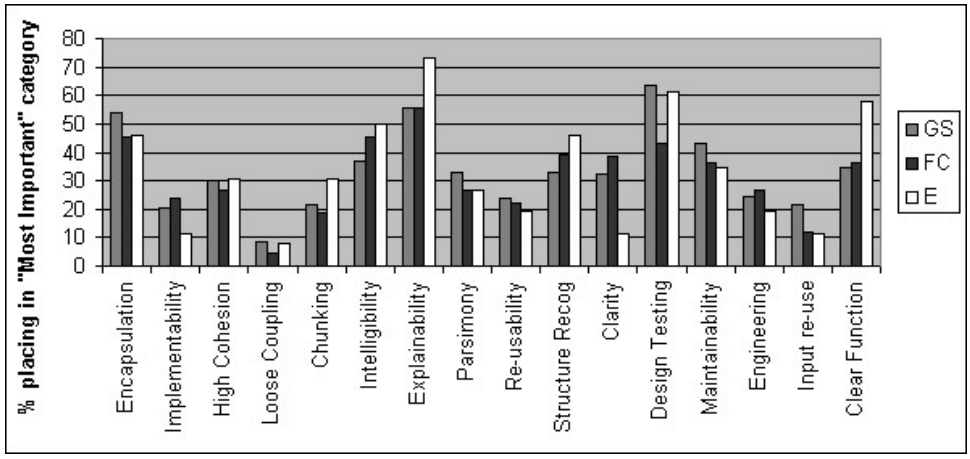


Figure 2: Most important, "current task" scenario

A number of χ^2 tests were run to determine if any apparent differences in frequencies were

statistically significant, across all scenarios for "most" and "least" priorotizations. In terms of between-group agreement, Educator criteria frequency counts were significantly different from those of the students for many of the individual criteria under at least one scenario. However, the frequencies of first competency and graduating students differed little, regardless of scenario. One of the surprising results is that the criteria *Loose Coupling* receives the fewest or second fewest number of counts for most important design criteria by FC students in every scenario. *Loose Coupling* also receives the fewest or second fewest number of counts for most important design criteria by GS students, except in the longevity scenario, where the difference in frequency with FC students for this criterion was the only statistically significant difference between the student groups across all criteria and all scenarios. So although this principle is sacrosanct for practicioners, it is rarely valued as such among students.

One of the things assessed is the degree to which individuals adjust the rankings of their design criteria when faced by different situations. Adams et al. (2003) observed that expert designers adapt the way they approach problems to match task constraints. The results of the present study indicate that students change their prioritizations when faced with different scenarios more than do educators, but there is no statistically significant difference in the amount of this change between the two student groups.

5 Conclusion

Each of the three analyses yielded results, with the main ones summarized here.

Design characteristics: there is a progression away from the textual and toward standard graphical notations with increases in education. The data also indicate that though a large number of students underestimate the importance of representing structural groupings and interactions between design parts, differences in institutional characteristics, including how software design is taught, might account for these differences.

Recognition of ambiguity: the percentage of both information gatherers and recognizers of ambiguity increases from first competency students to graduating students to educators. And those who recognize ambiguity or gather information had a higher success rate in addressing all requirements than those who did not. As with representation characteristics, there was considerable institutional difference in frequency of ambiguity recognizers and information gatherers.

Design criteria: there was little indication of changes to valuation of design criteria among students with differences in education. Each participant group changed its prioritizations in response to changing design contexts; surprisingly, students were more flexible, and adapted their criteria rankings to the context of the task to a greater degree than did educators.

Taken in total, these results suggest the following. First, that some design behaviors appear to be *developmental*, such as recognition of ambiguity and use of standardized design representations, in that there are increases in the occurrence of these behaviors with increases in educational attainment. Second, some design behaviors appear relatively invariant with respect to different levels of education within the Bachelor degree, such as design criteria valuation. It is possible that changes to these behaviors, such as appreciation of certain design criteria, is obtained primarily as a result of hard-won experience in "real-world" software development contexts. And third, some design behaviors are context-dependent, such as information gathering and representation of part-part interactions, suggesting that these behaviors are most amenable to changes in instruction.

6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0243242. Any opinions, findings, and conclusions or recommendations expressed in

this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thanks to Leigh Waguespack for assistance with statistical analysis and to Janet Rountree for assistance with data gathering and transcription.

References

Adams, R. S., Turns, J., Atman, C. J., 2003. What could design learning look like? In: Expertise in Design: Design Thinking Research Symposium 6. Sydney, Australia.

Atman, C. J., Chimka, J. R., Bursic, K. M., Nachtmann, H. L., 1999. A comparison of freshman and senior engineering design processes. Design Studies 20, 131–152.

Atman, C. J., Turns, J., Cardella, M., Adams, R. S., 2003. The design processes of engineering educators: Thick descriptions and potential implications. In: Expertise in Design: Design Thinking Research Symposium 6. Sydney, Australia.

Boehm, B., 1981. Software Engineering Economics. Prentice Hall.

Bruegge, B., Dutoit, A., 2000. Object-Oriented Software Engineering. Prentice Hall.

Christiaans, H. H. C., Dorst, K. H., 1992. Cognitive models in industrial design engineering. Design Theory and Methodology 42 (131-140).

CMM Correspondence Group, October 1997. Software product engineering (draft). Technical Report.

Fincher, S., Petre, M., Tenenberg, J., et al, September 2004. Cause for alarm?: A multi-national, multi-institutional study of student-generated software designs. Tech. Rep. 16-04, Computing Laboratory, University of Kent, Canterbury.
URL `http://www.cs.kent.ac.uk/pubs/2004/1953`

Goel, V., Pirolli, P., 1992. The structure of design problem spaces. Cognitive Science 16, 395–492.

Jeffries, R., Turner, A. A., Polson, P. G., Atwood, M. E., 1981. The processes involved in designing software. In: Anderson, J. (Ed.), Cognitive Skills and their Acquisition. Lawrence Erlbaum Associates.

Kaplan, S., Gruppen, L., Levanthal, L. M., Board, F., 1986. The components of expertise: a cross-disciplinary review. Tech. rep., University of Michigan.

McCracken, W. M., 2004. Research on learning to design software. In: Fincher, S., Petre, M. (Eds.), Computer Science Education Research. Routledge Falmer, Lisse, pp. 155–174.

McCracken, W. M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T., 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. SIGCSE Bulletin 33 (4), 125–180.

Petre, M., Fincher, S., Tenenberg, J., et al, June 2003. "My Criterion is: Is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. Tech. Rep. 6-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK.
URL `http://www.cs.kent.ac.uk/pubs/2003/1682`

Rowland, G., 1992. What do instructional designers actually do? Performance Improvement Quarterly 5 (2), 65–86.

Soloway, E., Ehrlich, K., 1984. Empirical studies of programming knowledge. IEEE Transactions on Software Engineering 10 (5), 595–609.

Automatic Assessment of Exercises for Algorithms and Data Structures – a Case Study with TRAKLA2

Mikko-Jussi Laakso and Tapio Salakoski
University of Turku
Department of Information Technology
Turku, Finland

Ari Korhonen and Lauri Malmi
Helsinki University of Technology
Department of Computer Science and Engineering
Espoo, Finland

{milaak,tapio.salakoski}@it.utu.fi,{archie,lma}@cs.hut.fi

Abstract

This paper presents the results of the case study introducing TRAKLA2 system in the course of data structures and algorithms at the University of Turku in 2004. We compared students' learning results with the results of the previous year. The numerical course results were clearly better than in 2003 when only pen-and-paper type exercises in classrooms were used. In addition, a survey was made with over 100 students on the changes in their attitudes towards web-based learning environments while getting acquainted with a wholly new system providing them automatic feedback and the option to resubmit their solutions. Our results show that such an on-line learning environment considerably increases positive attitudes towards web-based learning, and according to students' self-evaluations, the best learning results are achieved by combining traditional teaching and www-based learning.

1 Introduction

Automatic assessment (AA) tools for CS courses are being developed and gaining acceptance more and more widely at university level education. The survey of the ITiCSE working group "How shall we assess this" in 2003 indicated clearly that the experience of using AA tools correlates with a positive attitude towards applying such methods more widely, also when assessing higher order skills (Carter et al., 2003). The field where AA is most widely used is assessing programming exercises (*e.g.* Higgins et al. (2002); Luck and Joy (1999); Saikkonen et al. (2001); Vihtonen and Ageenko (2002)). Other applications include grading algorithm exercises (Bridgeman et al., 2000; Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000) and analyzing object-oriented designs and flowcharts (Higgins et al., 2002).

In this paper, we report the experiences on using the TRAKLA2 system for assessing algorithm exercises in which students simulate working of algorithms on a conceptual level. TRAKLA2 by Malmi et al. (2004); Korhonen et al. (2003) is a visual algorithm simulation exercise system that has been developed at Helsinki University of Technology (HUT). Students solve the exercises using graphical manipulation of conceptual visualizations of data structures on the screen. The system provides automatic formative and summative feedback on their work, and allows for resubmitting the solutions.

TRAKLA2 exercises were used for the first time in the basic data structures and algorithms courses at HUT in spring 2003. The system was used in parallel with the old TRAKLA system so that in total 14 TRAKLA2 exercises and 24 TRAKLA exercises were used in two courses[1]. In 2004, only TRAKLA2 was used and the total number of exercises was 26. During these two years more than 1000 students used the system.

In 2004, the University of Turku (UTU) also adopted TRAKLA2 for their data structure course with over 100 students. Compared with HUT this was a major cultural change on

[1] There were two versions of the course, one for CS majors and one for students of other engineering curricula.

the course. In HUT we have used automatically assessed algorithm simulation exercise since 1991 using the older TRAKLA tool, and thus the type of exercises and the culture of using automatic assessment is well-established both for the students and teachers. In UTU, however, no such exercises have been applied, except occasionally as pen-and-paper exercises without any automatic assessment.

In all these courses, both at HUT and UTU, TRAKLA2 exercises were a compulsory part of the course, and grading points achieved from the exercises had an effect on the final grade of the courses, although in slightly different ways. In HUT, TRAKLA2 exercises have an overall effect of 30% of the final course grade, whereas at UTU the TRAKLA2 exercises increased the number of examination points. In both institutes the minimum requirement was achieving at least 50% of the maximum points of the TRAKLA2 exercises.

The structure of the paper is the following. In the next section we give an overview of the TRAKLA2 system. Section 3 presents how the system was used in UTU, and how students attitudes and opinions were surveyed. Section 4 presents the results of the survey and final conclusions are included in Section 5.

2 Overview of the TRAKLA2 system

TRAKLA2 is a system for automatically assessing *visual algorithm simulation exercises* (Korhonen et al., 2003). It is based on the Matrix algorithm visualization, animation, and simulation framework (Korhonen and Malmi, 2002). TRAKLA2 distributes individually tailored tracing exercises to students and automatically assesses answers to the exercises. In visual algorithm simulation exercises, a learner directly manipulates the visual representation of the underlying data structures to which the algorithm is applied. The learner manipulates these real data structures through GUI operations with the purpose of performing the same changes on the data structures that the real algorithm would do. The answer to an exercise is a sequence of discrete states of data structures resulting from application of the algorithm, and the task is to determine the correct operations that will cause the transitions between each two consecutive states.

Let us consider the exercise in Figure 1. The learner has started to manipulate the visual representation of the Binary Heap data structure by invoking context-sensitive *drag-and-drop operations*. In the next step, for example, he or she can drag the key `C` from a *Stream of keys* into the left subtree of `R` in the binary heap. After that, the new key is sifted up via a *swap* with its parent until the parental dominance requirement is satisfied (the key at each node is smaller than or equal to the keys of its children). The swap operation is performed by dragging and dropping a key in the heap on top of another key. In addition, the exercise applet includes a push button for activating the Delete operation. The `Delete` button is applied in the second phase of the exercise to simulate the deleteMin operation. The student selects a node to be deleted and thereafter uses swap operations to heapify the tree again.

An exercise applet is initialized with proper *randomized input data*. The binary heap exercise, for example, is initialized with 15 alphabetic keys (Stream of keys), that do not contain duplicates. This means that the exercise can be initialized in more than 10^{19} different ways. The learner can *reset the exercise* by pressing the `Reset` button at any time. As a result the exercise is reinitialized with new random keys.

After attempting to solve the exercise, the learner can *review the answer* step by step using the `Backward` and `Forward` buttons. Moreover, the learner can *ask feedback* on his or her solution by pressing the `Grade` button in which case the learner's answer is checked and immediate feedback is delivered. The feedback reports the number of correct steps out of the total number of required steps in the exercise. Finally, it is possible for the student to *submit the answer* to the course database using the `Submit` button. By default an answer to an exercise can be submitted unlimited times; however, a solution for a specific instance of exercise with certain input data can be submitted only once. In order to resubmit a solution

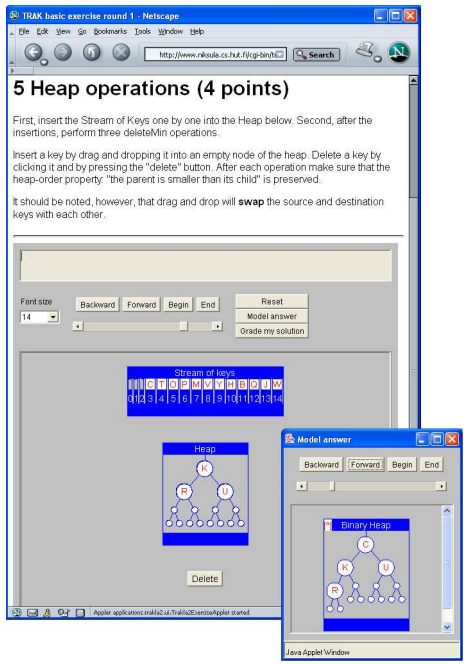


Figure 1: TRAKLA2 applet page and the model solution window.

to the exercise, the learner has to reset the exercise and start over with new randomized input data. Thus, it is not possible to grade the same solution and improve it arbitrarily before submitting it.

A learner can also *examine the model solution* of an exercise. It is represented as an algorithm animation so that the execution of the algorithm is visualized step by step. In Figure 1, the model solution window is opened in the front. The states of the model solution can be browsed using the `Backward` and `Forward` buttons. For obvious reasons, after opening the model solution for given input data, a student cannot submit a solution until the exercise has been reset and resolved with new random data.

Each TRAKLA2 exercise page (*e.g.*, Fig. 1) consists of a description of the exercise, an interactive Java applet, and links to other pages that introduce the theory and examples of the algorithm in question. The current exercise set covers almost 30 assignments on basic data structures, sorting, searching, hashing, and graph algorithms. Appendix A lists the current exercises in TRAKLA2.

3 Algorithms and data structures course at University of Turku

Algorithms and data structures (DSA-UTU) course at University of Turku included 56 lecture hours, 10 classroom exercises (each 2 hours) and 22 TRAKLA2 exercises in spring 2004. Previous courses were held with 56 lecture hours and 13 classroom exercises (2 hours each). The classroom exercises consist of five single exercises like illustrating exercises, proofing exercises, *etc.* TRAKLA2 exercises, however, are most effective to represent exercises in which the task is to illustrate how a specific algorithm works with given input values. Thus, the number of classroom exercises was cut down after TRAKLA2 was taken in use. In numbers, classroom exercises decreased from 65 to 50. Each TRAKLA2 exercise was given points from one to four. There was a possibility to get in total of 47 TRAKLA2 points in DSA-UTU course. The TRAKLA2 exercises were divided into three rounds by synchronizing the exercises to topics in hand in the DSA-UTU course.

3.1 Grading and requirements of the DSA-UTU course

There were two ways of passing the course. By taking the final examination (0-32 course points) or by taking two midterm-examinations (both 0-16 course points). In either way, student must still fulfill the minimum requirements, which are: i) students must do at least 20 of the 50 classroom exercises, ii) students must get at least 50% of the TRAKLA2 points (maximum 47 points), and iii) students must get at least 20 course points out of the total of 40 course points in share.

It was possible to get 32 course points from the examination(s) and eight course points from TRAKLA2 exercises. Conversion of TRAKLA2 points to course points was linear between the minimum requirements 50% (pass with zero course points) and 100% TRAKLA2 points (8 course points that is 20% of the maximum).

In comparison with earlier DSA-UTU courses, TRAKLA2 exercises replaced one question in the examination or a half of a question in both midterm-examinations. Traditionally one of the five questions in examination has been such an illustrative type of assignment, and this was the very question now replaced by TRAKLA2 exercises.

The final grading of the DSA-UTU course was in scale from one to three with 0.25 steps. By getting 20 course points the student will get lowest grade, which is one. In addition, by doing 60% or 80% of classroom exercises, any student can get an additional + or $\frac{1}{2}$ to his grade, respectively (still requires the student to fulfill the course minimum requirements).

3.2 The setting of the study

The attitudes of the students in UTU where studied using questionaries. Three sets of questionaries where filled by the students during the course. The first questionary at the beginning of the course, the second (Mid) at the first midterm-examination (after the first round of TRAKLA2 exercises), and the third one at the second midterm-examination (after the courses).

The first questionary was aimed to gather information about students' attitudes towards and experiences of www-based materials and tools in earlier courses. Questions also covered students' opinions about how well www-based exercises are suitable in DSA-UTU course (scale in numbers 1-5, 5 is the best). It was also asked how students prefer to do DSA-UTU courses exercises (by www-exercises, by classroom-exercises, or mixed). Students ranked different ways of doing exercises in order from one to three (one is the best, three is the worst) by their own interest. In the same way, the students also self assessed the level of their learning.

There were two main questions of yes-no type in the second questionary. The first question was about the contribution of TRAKLA2 system in the learning of course topics. The second question was about usability of TRAKLA2 and about any problems of using it. Both questions included also possibility of free text comments.

On the third questionary, the questions on the first and second questionaries were repeated. In addition, further comments and suggestions were asked for.

4 Results and discussion

As a whole, the TRAKLA2 system has worked well with surprisingly good results both at HUT and at UTU. In 2004, 30% of the students at HUT achieved the maximum number of points for the 26 exercises, and over 55% achieved 90% of the maximum. Only 15% of the students failed to get the required minimum of 50% of the points; in practice these were students who dropped the whole course early. At UTU the results were even better. The average number of points achieved was 7.34 points out of maximum 8 points.

Students' opinions of the system were determined through a web-based survey at the end of the HUT course in 2003. 364 students answered. 80% of them gave an overall grade of 4 or 5 to the system in scale 0–5, where 5 was the best grade. The system was almost unanimously

considered to aid learning and easy to use. In UTU, free feedback from the system was well in line with these observations. In addition, a different questionnaire was carried out which surveyed how students' attitudes towards on-line learning environments was changed when they had used TRAKLA2. This pointed out clearly that the attitudes became more positive.

In the following, we present a more detailed analysis of the results of the survey on the UTU students' opinions and attitudes towards www-based learning. Moreover, the learning results are presented based on students' self evaluation. After that, results derived from course statistics are presented, including the numbers of students failed/passed in total, average grades, attendances in classroom and TRAKLA2 exercises, *etc.* The data is compared with the data from DSA-UTU course in spring 2003, when the course was given by the same lecturer and the classroom exercises were very similar to those in spring 2004.

4.1 The survey results

There were 96 students answers to the first questionary ('Start'), 103 to the second ('Mid'), and 81 to the third questionary ('End'). At the Start and End the students were asked about their opinion on the suitability of www-based exercises for learning data structures and algorithms. The answering alternatives were well (5), quite well (4), neutral (3), quite bad (2), and bad (1). The Start average were quite high, 3.94, and the End average were even higher, 4.84. These results indicate that www-based exercises are very suitable for learning data structures and algorithms. Also the increase of the average during the course is large and therefore it seems that www-based exercises were well accepted and approved even by students without strong positive prejudice.

As to the qualitative analysis, also the free text comments were analyzed. There were a number of answers in which students said that it is much more elegant to do this kind of illustrative type of exercises with TRAKLA2 rather than doing the same in a piece of paper step by step. Also, it was often mentioned that TRAKLA2 exercises concretized the actions and operations of an algorithm. It was also confirmed that the immediate feedback by the TRAKLA2 system helped the students to find the point where they made a mistake and encouraged them to further deepen their understanding of the subject. This is also reflected by course statistics.

In the Mid and End questionaries, the students were asked how TRAKLA2 exercises contributed to their learning. In the Mid, the question was formulated as yes/no-type, and 94% of students answered that TRAKLA2 exercises did aid their learning process. At the End, the students were asked to describe the contribution on a scale from 1 to 5 (5 is the best). The average of the answers was 4.10, and 84% of the students selected 4 or 5, while there were only two answers below 3. This result is well in line with previous results from the study at HUT.

We also asked the students to give their preference on the three ways of doing exercises: traditional classroom exercises, web-based exercises, or mixed (see Figure 2). In the same manner, the students were asked to assess the level of their learning (Figure 3). It can be seen from the answers that the students' attitudes changed positively towards www-based exercises during the course. Students prefer the most to do exercises by combining traditional and www-based exercises even in the starting questionary, and their opinion strengthened during the course so that at the end, nearly three out of four students considered mixed exercises the best. The same happened to the students' self assessment of their learning. The mixed alternative is clearly the most suitable way to learn data structures and algorithms. Furthermore, if the students' were to choose only between traditional and web-based exercises, they would prefer traditional over www-based exercises due to their better contribution to learning. This is very interesting result suggesting that although web-based exercises complement very well traditional classroom exercises, the former can hardly replace the latter in general.

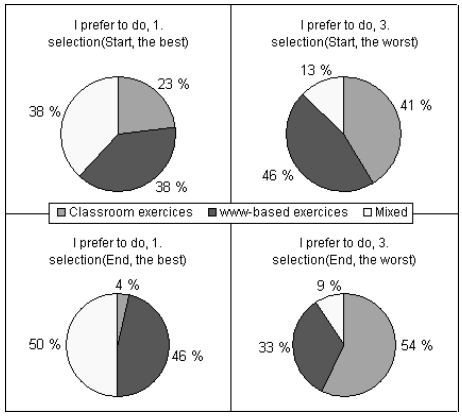


Figure 2: I prefer to do

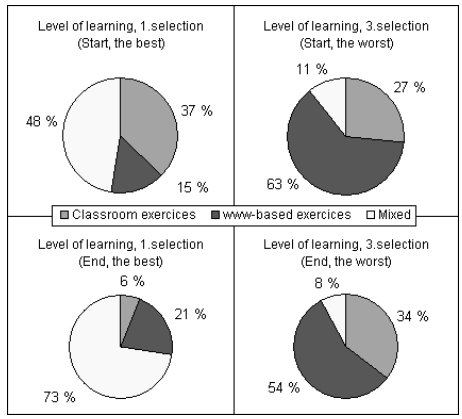


Figure 3: The level of learning

Table 1: Students' activity in classroom exercises

	Spring 2003	Spring 2004
Number of (#) attendants	186	165
Average % of classroom exercises (only who did at least 40%)	54.5	60.3
Number of (#) attendants who did 0% - 40% of classroom exercises(failed)	76	43
# attendants who did 40% - 60% of classroom exercises(no bonus)	80	79
# attendants who did 60% - 80% and received + from classroom exercises to their final grade	18	21
# attendants who did 80% - 100% and received $\frac{1}{2}$ from classroom exercises to their final grade	12	22

4.2 The course statistics

Table 1 shows statistics about students' activity in classroom exercises from DSA-UTU courses in spring 2003 and spring 2004. In addition, students got as an average of 7.34 course points from TRAKLA2 exercises, and 69,2 % of students did 100 % of TRAKLA2 exercises.

As we can see from the statistics, in spring 2004, the students were more active not only in using TRAKLA2 but also in other part of the course compared with 2003; especially, the average performance in classroom exercises raised from 54,5% to 60,3%. There is also a statistically significant difference (χ^2-test, $p < 0.01$) between the two years in the statistics in Table 1. Thus, a larger number of students received additional + / $\frac{1}{2}$ to their final grade in 2004 than in 2003. These observations confirm that the introduction of TRAKLA2 system enhanced the students' motivation and performance on the DSA-UTU course.

In Table 2, there are shown the basic statistics from DSA-UTU courses in 2003 and 2004,

Table 2: Course statistics

	Spring 2003	Spring 2004
Number of (#) attendants	186	165
Average course points	26.15	27.51
Average of the final grades	2.01	1.97
# attendants who were in second midterm-examination	58	82
# passed attendants	49	81
% of attendants who were in second midterm-examination and passed the course	84.5	98.7

which were of about the same size. There was a major increase in number of passed attendants. On the other hand, when looking at the average of course points (t-test, $p = 0.19$) and the average of final grades (χ^2-test, $p = 0.12$), there is no statistically significant difference between those two courses. Combining these two observations it can be concluded that TRAKLA2 aided many students to get over the edge and pass DSA-UTU course. Hence, it seems that TRAKLA2 is truly useful for those students who have difficulties learning data structures and algorithms by classroom exercises.

5 Conclusions

The study showed that students' attitudes strengthened positively towards www-based exercises. Moreover, the mixed alternative is far the most appropriate way to learn topics of DSA course, and it's well approved and preferred by students. Furthermore, the results suggest that www-based exercises constitute a good amendment to DSA course. However, it seems also that there exits a certain desire for more traditional exercises. Whether these students' exceptations can be fulfilled by a future version of TRAKLA2 or similar web based tools, remains an interesting challenge.

Interface of the TRAKLA2 system was easy to use, and features like possibility to get immediate feedback and the resubmit alternative aided students to complete given exercises, and by that they enhanced their learning. In addition, the study pointed out that the TRAKLA2 system affected positively on students' behaviour on other areas of DSA-UTU course, and an average student did more work for learning the course's topics. In the same time, the number of passed attendants raised from 49 to 81, thus the TRAKLA2 system aided especially less talented students to get over the edge and pass the course.

At this time, the only existing type of TRAKLA2 exercise is to illustrate how a specific algorithm works on given input. Basically, this calls for tracing the execution of the algorithm, whereas the system currently offers no support for constructive exercises, such as in which a problem is described, example input and output values are given, and the task is to construct the algorithm.

In conclusion, the TRAKLA2 system was well accepted and approved by students, and it will be used in forthcoming DSA courses also at UTU. A key task of the future is to develop novel types of TRAKLA2 exercises in collaboration between Helsinki University of Technology and University of Turku.

References

Bridgeman, S., Goodrich, M. T., Kobourov, S. G., Tamassia, R., 2000. PILOT: An interactive tool for learning and grading. In: The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education. ACM, pp. 139–143.
URL `citeseer.nj.nec.com/bridgeman00pilot.html`

Carter, J., English, J., Ala-Mutka, K., Dick, M., Fone, W., Fuller, U., Sheard, J., 2003. ITICSE working group report: How shall we assess this? SIGCSE Bulletin 35 (4), 107–123.

Higgins, C., Symeonidis, P., Tsintsifas, A., 2002. The marking system for CourseMaster. In: Proceedings of the 7th annual conference on Innovation and technology in computer science education. ACM Press, pp. 46–50.

Hyvönen, J., Malmi, L., 1993. TRAKLA – a system for teaching algorithms using email and a graphical editor. In: Proceedings of HYPERMEDIA in Vaasa. pp. 141–147.

Korhonen, A., Malmi, L., 2000. Algorithm simulation with automatic assessment. In: Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education. ACM, Helsinki, Finland, pp. 160–163.

Korhonen, A., Malmi, L., May 2002. Matrix — Concept animation and algorithm simulation system. In: Proceedings of the Working Conference on Advanced Visual Interfaces. ACM, Trento, Italy, pp. 109–114.

Korhonen, A., Malmi, L., Silvasti, P., Nikander, J., Tenhunen, P., Mård, P., Salonen, H., Karavirta, V., 2003. TRAKLA2. URL: http://www.cs.hut.fi/Research/TRAKLA2/ (27.9.2003).

Luck, M., Joy, M., 1999. A secure on-line submission system. Software - Practice and Experience 29 (8), 721–740.

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppl, O., Silvasti, P., 2004. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. Informatics in Education 3 (2), 267 – 288.

Saikkonen, R., Malmi, L., Korhonen, A., 2001. Fully automatic assessment of programming exercises. In: Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01. ACM, Canterbury, United Kingdom, pp. 133–136.

Vihtonen, E., Ageenko, E., 2002. Viope-computer supported environment for learning programming languages. In: The Proceedings of Int. Symposium on Technologies of Information and Communication in Education for Engineering and Industry (TICE2002). Lyon, France, pp. 371–372.

A TRAKLA2 Exercises

Table 3: The visual algorithm simulation exercises in TRAKLA2 system. The column *name* describes the topic and the *description* characterizes the exercise. The roman numbers (i-iv) indicate the separate exercises and the number of sub-topics.

Name	Description
Insertion into (i) Binary search tree, (ii) Digital search tree, and (iii) Radix search trie	The learner is to insert random keys into an initially empty search tree by dragging and dropping them into the correct positions.
Binary search tree deletion	The learner is to remove 4 keys from a binary search tree of 15 to 20 keys. Pointer operations are simulated by directly manipulating the edges that connect the nodes of the tree in the visualization.
Faulty Binary Search Tree	The learner is to show how to bring the following binary search tree in an inconsistent state: duplicates are allowed and inserted into the left branch of an equal key, but the deletion of a non-leaf node relabels the node as its successor.
AVL tree (i) insertion, (ii) single rotation, and (iii) double rotation	The learner is to (i) insert 13 random keys into an initially empty AVL-tree. The tree (i-iii) has to be balanced by rotations. The rotation exercises (ii-iii) require pointer manipulation, while the insertion exercise (i) provides push buttons to perform the proper rotation at the selected node.
Red-black-tree insertion	The learner is to insert 10 random keys into an initially empty Red-Black-tree. The color of the nodes must be updated and the tree must be balanced by rotations.
BuildHeap algorithm	The learner is to simulate the linear time buildheap algorithm on 15 random keys.
Binary heap insertion and delete min	The learner is to a) insert 15 random keys into a binary heap and b) perform three deleteMin operations while preserving the heap order property (see Fig. 1).
Sequential search: (i) Binary search, and (ii) Interpolation search	The task is to show which indices the algorithm visits in the given array of 30 keys by indicating the corresponding keys.
Tree traversal algorithms: (i) preorder, (ii) inorder, (iii) postorder, and (iv) level order	The learner is to show which keys in a tree the algorithm visits by indicating the visited keys in the required order.
Preorder tree traversal with stack	The learner is to simulate how the stack grows and shrinks during the execution of the preorder algorithm on a given binary tree.
Fundamental Graph algorithms: (i) Depth First Search, and (ii) Breadth First Search	The learner is to visit the nodes in the given graph in DFS, and BFS order.
Minimum spanning tree algorithms: Prim's algorithm	The learner is to add the edges into the minimum spanning tree in the order that Prim's algorithm would do.
Shortest path algorithms: Dijkstra's algorithm	The learner is to add the edged to the shortest paths tree in the order that Dijkstra's algorithm would do.
Open addressing methods for hash tables: (i) linear probing, (ii) quadratic probing, and (iii) double hashing	The learner is to hash a set of keys (10-17) into the hash table of size 19.
Sorting algorithms: (i) Quicksort, and (ii) Radix Exchange sort	The learner is to sort the target array using the given algorithm.

Cognitive skills of experienced software developer: Delphi study

Sami Surakka and Lauri Malmi
Helsinki University of Technology

sami.surakka@hut.fi and lauri.malmi@hut.fi

Abstract

In this paper a qualitative study of cognitive skills of experienced software developers is presented. The data for the study was gathered using the Delphi method. The respondents were 11 software developers who have worked at least five years after their graduation. The respondents were found using recommendations since the goal was to find especially good software developers. Thus, they are not a statistically representative sample from all software developers but more like a focus group. Two questionnaire rounds were conducted. In the first round, the respondents mentioned altogether 32 different skills. In the second round, 10 of the respondents answered and evaluated the importance of these 32 skills. The results are divided into two categories: composition and comprehension. For each skill, the evaluated degree of difficulty of the skill is presented (e.g., does the skill efficiently differentiate experts from novices).

1 Introduction

What are cognitive skills? According to ERIC Thesaurus (2004), the term *thinking skills* should be used for the term *cognitive skills*. The description for the term thinking skills is the following:

> Interrelated, generally "higher-order" cognitive skills that enable human beings to comprehend experiences and information, apply knowledge, express complex concepts, make decisions, criticize and revise unsuitable constructs, and solve problems—used frequently for a cognitive approach to learning that views explicit "thinking skills" at the teachable level.

In this study the goal has been to identify cognitive skills that are important for expert software developers' work. Our research origins from the need to better understand what kind of topics and skills should be included in the Masters level education of software systems specialists in the Helsinki University of Technology. Typical sources for such curriculum development work include various model curriculums such as Computing Curricula 2001 (Engel and Roberts, 2001). However, they mostly concentrate on listing topics to be covered in the curriculum. The skills to be achieved during the education are covered more vaguely. Since programming is a high-level cognitive skill, we wanted to find out in some more detail what kind of cognitive skills should be trained in the education.

We decided to search for high-level software development experts and ask from them which topics in computer science they consider important for their work. Moreover, we were interested in identifying tacit knowledge needed in software development. Since such information is difficult to be grasped with simple questionnaires we decided to apply the Delphi method (Wilhelm, 2001) in which people in the same focus group are queried two or more times. After each time a summary of results is presented for them followed by more closely defined questions of the topic of interest. Delphi is a qualitative research method, where the quality rather than the number of respondents is the more important factor. The statistical reliability of the results is therefore not the general goal, and thus the number of respondents need not be very large. In this study we selected, based on some general quality criteria, 11 respondents among a group of recommended 59 experts. Two questionnaire rounds were

performed, and the second round concentrated especially on the tacit knowledge of software development. In this paper we concentrate on the results of the second questionnaire round.

The structure of the paper is the following. First, we consider some related work in Section 2. In Section 3 we describe the research method in some detail. The results are presented and analyzed in Section 4. A discussion including some implications to education and evaluation of this research summarizes the paper.

2 Related work

We did not find any research papers where the Delphi method has been used in the field of psychology of programming. This is understandable because it is not common to use even questionnaires as a research instrument in this field.[1] Because the lack of similar research, some more general references are presented next. In the end of this section it is explained how these issues relate to our research.

Greeno and Simon (1988) wrote "Computer programming may be characterized 'as a whole' as a design task." Brooks (1983) wrote about design task domains:

> ..., two fundamental activities in design task domains are composition and comprehension. Composition is the development of a design and comprehension results in an understanding of a design. The essence of the composition task in programming is to map a description of what the program is to accomplish, in the language of real-world problem domains, into a detailed list of instructions to the computer designating exactly how to accomplish those goals in the programming language domain. Comprehension of a program may be viewed as the reverse series of transformations from how to what.

Stanislaw et al. (1994) divided expertise in computer programming into two components that were time-based expertise and multiskilling expertise. They wrote (p. 351): "*Time-based* expertise corresponds to the conventional notion of expertise, and is a function solely of the time spent on programming. *Multiskilling* expertise, by contrast, accrues through exposure to a variety of programming languages and tasks, and is related to the cognitive development of higher-level programming schemata." Detienne (2002, p. 35) wrote that one of the characteristics that distinguishes "super experts" or "exceptional designers" from other experts is: "a broader rather than longer experience: the number of projects in which they have been involved, the number and variety of the programming languages they know."

In addition, Detienne (2002, p. 35) wrote that experts carry out some aspects of programming task completely automatically. She refereed to Wiedenbeck (1985, p. 383) who found that experts were faster and did fewer mistakes than novices when both groups had to do a series of timed true/false decisions about short, textbook-type program segments. One might assume that, for example, the following skills are automated gradually when the programming experience increases: (a) using basic commands of an editor (such as Emacs) and the programming system frequently used, and (b) knowing details of syntax and code conventions of a certain programming language such as C.

The previous issues relate to this study as follows: (a) We have used the two activities, composition and comprehension, to interpret and divide our results. (b) The division time-based expertise vs. multi-skilling expertise was used so that we required that at least half of the respondents should be characterized as multi-skilled experts. (c) The concept of skill automation was used with the questions about cognitive skills: the first question concerned higher-level skills and the second question concerned skills that might be partially or totally automated.

[1] We found only seven articles where questionnaire has been used, for example (Capretz, 2003). However, none of these articles is really related to our study beside the use of questionnaires.

3 Method

An overview of the Delphi method can be found, for example, from (Wilhelm, 2001). The method was originally used to forecast the future; the name originates from "the oracles of Delphi" where Delphi refers to an ancient Greek island. However, in this study, estimating future was only a small part.

Some basic properties of the method are the following. First, there are several questionnaire rounds. Second, the results from the previous round are used as material for the next round. Thus the respondents may change or tune their previous answers. One of the main reasons for using Delphi was that it allows group communication without gathering all respondents to the same place in the same time, which in this case would have been very difficult to achieve. Moreover, in this way the respondents had more time to consider their answers and make their views more explicit.

Originally consensus building has been an important part of the Delphi method. In this research, however, the second questionnaire round was not used for building consensus on the whole issue but targeted more to refining the results of an interesting part of the first questionnaire; that is, cognitive skills. The first questionnaire had three open questions about cognitive skills required by a software specialist. Based on the answers in total 36 different skills were identified. In the second round the respondents defined the level of these skills, that is, how long learning and experience is needed before such a skill is mastered. The questionnaires are presented in more detail in Section 3.2.

The decision of limiting the second questionnaire to only one area of interest was based on several reasons: (a) The results from the other areas of the first questionnaire were satisfactory enough. Thus, the need to conduct a second questionnaire round for the sake of the other areas was low, (b) The respondents thought that the questions about cognitive skills were the most difficult to answer. We interpreted this as a hint to explore more this area, (c) Regardless of the answering difficulties, some respondents thought cognitive skills as interesting or promising area for this kind of study. This was our own opinion, as well, and finally, (d) In the beginning of the study we promised to the respondents that participating would take 1-3 hours, and we wished not to break this promise.

After the cognitive skills were chosen as the topic for the second questionnaire round, the goal was set to evaluate how demanding or difficult the different cognitive skills that were mentioned during the first round are.

3.1 Finding respondents

The goal was to find 10-20 especially good software developers. The respondents were found using recommendations. Thus, they are not a statistically representative sample from all software developers but more like a focus group. Probabilistic sampling was not used because it was difficult to identify the target group using properties such as age, education, and title. For example, the title and working years are not enough to separate especially good software developers from poor or intermediate developers. Our decision thus fits well with guidelines presented by Kitchenham and Pfleeger (2002, p. 19): "Nevertheless, there are three reasons for using non-probability samples: 1. The target population is hard to identify. For example, if we want to survey software hackers, they may be difficult to find..."

The minimum criteria were a degree, five years working experience after graduation, at least half of time used to programming during these five years, and at least 100,000 lines of self implemented code. In addition, at least half of the respondents should have versatile software development experience. Here, versatile means different kind of projects, for example various programming languages and application domains. Two extra criteria were that (a) maximum of three respondents can be included from the same organization and (b) only one respondent can work full-time at the Helsinki University of Technology, where the authors work themselves. The degree could be from other programs than computer science and engineering.

For example, some older respondents had the degree from electrical engineering. The title of the respondent needed not be programmer, software developer or software engineer, since the important issue was only that their work included enough programming.

Altogether, 59 persons were recommended. 40 of them were not asked because of several different reasons (e.g., the person was graduated less than five years ago). Thus, 19 persons were asked to participate starting from those who had more recommendations. From these 19 persons, 11 promised to participate.

The criterion of at least 100,000 lines of self-implemented code and enough programming experience during the last five years were checked when the person was asked to take part. Some candidates declined because of these two conditions. The criterion of at least half of the respondents should have versatile software development experience was controlled with the first questionnaire. No respondents were excluded because of this criterion.

3.2 Questionnaire rounds

Two questionnaire rounds were conducted. The first questionnaire was answered between November 2003 and January 2004, the second questionnaire between January and February 2004. During the first round, most respondents answered so that they were able to ask questions from the researcher (from one of us) who was present during they answered. The researcher was not present during answering on the second round. The mean answering time for the first round was one hour and six minutes, and 54 minutes for the second round. The original questionnaires are available in Finnish only at (Surakka, 2004). However, their main properties are presented in the following two subsections.

3.2.1 First questionnaire

The first questionnaire had 14 open questions and 14 multiple-choice questions. The topics were (a) background information from the respondent, (b) the importance of various subjects and skills for software development, such as discrete mathematics and concurrent programming, (c) cognitive skills, (d) problem solving techniques, and (e) software quality. For brevity, only results about the background information and cognitive skills are presented in this article.

The questions about background information were title, proportion of time used to programming, number of employees under the respondent, lines of code implemented by the respondent, number of different groups involved, number of different projects, personal skills in various subjects (42 subitems such as discrete mathematics and object-oriented programming), skills in various programming languages and knowledge of various operating systems.

Instead of cognitive skills, the term “tacit knowledge” was used because we assumed that it would be easier to understand for the respondents. An explanation of the concept including initial division to cognitive skills and technical skills was given before the questions. The three questions were:

- For top-level software developer, what are important mental models, beliefs and understanding that belong to the cognitive element of tacit knowledge?
- For top-level software developer, what topics or skills belong to the technical element of tacit knowledge? This can also be called as skills that are located in the fingertips.
- Do you believe that some area of tacit knowledge will be more important in the future?

3.2.2 Second questionnaire

The second questionnaire was based on the respondents’ answers and comments to the first questionnaire. These were analyzed to identify and separate different skills mentioned in the comments. Comments clearly denoting the same skill were joined. Typing skill was included

into the list, based on researcher's observations, even though the respondents did not mention it. Finally we had a list of 36 comments each identifying at least one skill, for the next round.

In the second questionnaire, the respondents had to evaluate the level of these comments according to the following categories:

1. Very low-level skill that even novices can learn quickly (during a 1–4 credits basic course)
2. Somewhat low-level skill that requires working experience of 3–6 months to be learned, for example
3. Somewhat high-level skill that starts to differentiate good programmers from less good programmers
4. Very high-level skill that takes usually several years to learn and typically only top-level programmers have this skill.

The second questionnaire also had questions about problem solving techniques, use of editor, and typing skills. For brevity, these results are not reported in this article.

4 Results

First, some background information about respondents is presented. Second, the results about respondents' opinions from cognitive skills are presented.

4.1 Background information of respondents

All respondents were male and mean of respondents' ages was 37.1 years. Their degrees were as follows: one college degree in computer science and engineering (9%), five masters in computer science and engineering (45%), three masters in other engineering disciplines (27%), one doctor from applied mathematics (9%) and one doctor from computer science and engineering (9%). The respondents' positions were distributed into following groups: senior software engineers and developers 45%, researchers 27%, and managers or directors 27%.

Each respondent was asked to give himself a grade in 42 subjects or skills related to various fields of computer science, or other sciences (mathematics, physics), and software development phases of the waterfall model. In Table 1 are shown the ten subjects or skills that respondents evaluated they knew best on average. There are two issues that are worth noticing. First, script programming skills are ranked very high. This obviously correlates with the heavy use of Unix/Linux environment in their work. We did not ask more questions on scripting on the second round. However, our interpretation of this phenomenon is that for this target group scripting is a regular method for solving simple computational problems, for example, filtering and manipulating data files, or building auxiliary tools for them. This is strongly related with the important cognitive skills of recognizing the need for building new tools and choosing a suitable tool for each purpose.

The second observation is that functional programming is ranked much higher than the general use of functional programming languages in software production would indicate. We believe that this is related to multi-skilling. A plausible explanation is that many of the respondents have used functional programming during the career and/or hobby programming. Based on answers to the open question about working experience, at least four (36%) respondents had actually used Lisp in some work project.[2]

[2]Nine (82%) respondents have graduated from the Helsinki University of Technology where Scheme was the language of the first compulsory programming course in the degree program of computer science and engineering (CSE) during 1989-2003. However, this is not a suitable explanation because all these nine respondents were admitted before 1989 or were from other degree programs than CSE. That is, the course in question was not compulsory for them.

Table 1: Respondents' top strengths according to question "Give yourself a grade in the following subjects or skills" (scale: 1 poor ... 4 excellent).

Rank	Subject or skill	Mean
1	Implementation	3.8
	Procedural programming	3.8
3	Data structures and algorithms	3.5
	Script programming	3.5
5	Design	3.4
	Object-oriented programming	3.4
7	Operating systems	3.1
	Testing	3.1
	Version and configuration management	3.1
10	Functional programming	3.0

4.2 Respondents opinions about cognitive skills

In the second questionnaire, the statements of skills were divided according to the division used in the first questionnaire. However, for this article we reclassified the results into two categories: composition and comprehension. We also combined some comments. Two comments are not presented in the tables because they are not related only to software development. These two comments and their means were Being systematic 2.1 and Ability to type using ten fingers 2.1. Thus, the tables contain fewer comments than the second questionnaire did.

First, the results related to composition are presented in Table 2. The comments are ordered according to the means. The numbers in the leftmost column are used for commenting the items.

Even though statistical analysis was not our main purpose, we were curious to see, whether the observed differences are significant or not. We used the Mann-Whitney test (Conover, 1999, pp. 271-275) for the analysis because this nonparametric test is suitable for small samples. *Note that the test compares the ranks, not the means.* However, for brevity we present the test results in the same column with the means. The ranks of single items were compared to the ranks of all items. A star (*) indicates that the difference is statistically significant ($p<0.01$). If the star is missing, the difference is not statistically significant.

In Table 2, there are a few observations which need commenting. First, the high mean of item "2a Automating one's own work using scripts, keyboard macros etc." obviously does not indicate the time needed to learn such skills. Instead, it indicates the time needed to use them efficiently as one's personal tools, when necessary. Our assumption is that this is a skill which is analogous to bottom-up software design, where the programmer recognizes the need for general-purpose procedures and data structures. Thus, it has a role in differentiating excellent developers from others. Second, the items "Design of interfaces" and "Isolating the implementation behind well defined (and documented) interfaces" are kept separate. The first one is more associated with *designing* and the latter one with *using* interfaces. It is obviously easier to learn to use ready-made interfaces properly than actually designing interfaces that support good software architecture.

Third, comments 2b and 7b are similar but we think that 2b is broader than 7b. Comment 2b includes also low-level knowledge, for example knowing language's keywords by heart. Forth, we think that the low ranked items 15a and 17 are not really cognitive skills, but other kind skills or knowledge. However, we have not omitted these items from the table because they are related to composition.

In Table 3 we present the results related to category "Comprehension". As a general

Table 2: Comments classified into category "Composition": Means to question "What do you think is the level of this skill?" Scale was: 1 very low-level skill. . . 4 very high-level skill.

Number	Comment	Mean
1	A good programmer has always a model. The code itself comes from spine and brains operate only the model.	3.6*
2a	Automating ones own work using scripts, keyboard macros etc.	3.5*
2b	Mastery of a certain programming language or a certain environment	3.5*
4	Writing code so well that it is not even necessary to comment	3.4
5	Design of interfaces	3.3
6	Choosing as optimal data structures and algorithms as possible	3.1
7a	Ability to find right abstractions	3.0
7b	Mastery of the structures and idioms that are characteristic for each language or environment	3.0
9	Ability to write code clearly and shortly	2.9
10a	Choice of the programming language	2.8
10b	Implementing programs as independent from the operating environment as possible	2.8
12	Isolating the implementation behind well-defined (and documented) interfaces	2.7
13	Changing lower level cognitive models/design patterns to code. For example, table field in C/C++ object and its memory management get/set/constr/destr.	2.6
14	Identifying concepts	2.4
15a	Ability to find existing Open Source solutions from Net and being familiar with libraries	2.3
15b	Procedural or object-oriented way of thinking about programming	2.3
17	Documenting code	1.9*

A star (*) indicates that the difference is statistically significant ($p<0.01$).

note, it is interesting that the respondents have used often words like "see" and "notice" to describe these skills. We think that item "13 Understanding the function of programming languages and computer (e.g., parameter passing, order of execution, and concurrency)" is rather explicit than tacit knowledge.

5 Discussion

In this section the research is evaluated, conclusions are drawn, implications to education are presented, and possibilities for future research are considered.

5.1 Evaluation of the research

This study would have been very different if the original main goal was to gather information from cognitive skills of software developers. Questionnaires are used seldom in psychology of programming where experimental research setting is dominant. One source of criticism is that questionnaires measure opinions, not observable behavior. However, in this research the purpose was to measure especially the opinions of experts. In addition, we think that Delphi method was suitable for this type of research. The follow-up questionnaire round was necessary to describe more explicitly tacit knowledge that is more or less vague concept.

During the first questionnaire round, most respondents commented that the questions

Table 3: Comments classified into category "Comprehension": Means to question "What do you think is the level of this skill?" Scale was: 1 very low-level skill ... 4 very high-level skill.

Number	Comment	Mean
1	Ability to see all possible alternatives from the source code (this comment was related to debugging)	3.9*
2	Ability to notice isomorfisms with some known problem	3.6
3	Ability to evaluate how the system will operate even before its implementation has been started	3.5
4a	Ability to see esthetic values in solutions	3.4
4c	Ability to see the big picture. What is the core of the problem and how it is connected to the environment around it?	3.4
6a	Ability to distinguish essential matters	3.2
6b	Interpreting the program as whole	3.2
8a	Ability to change fluently - abstraction level (e.g., single line of code vs. procedure or big picture vs. details), - perspective (e.g., is the control flow or the data flow of the program examined), - concepts (e.g., are the concepts of program or the concepts of application domain considered) - and view (e.g., users needs vs. maintenance vs. development speed).	3.1
8b	Ability to debug	3.1
10	Ability to see symmetries	3.0
11	Exploring the architecture of the existing systems	2.9
12	Ability to see a big problem as several partial problems	2.7
13	Understanding the functioning of programming languages and computer (e.g., parameter passing, order of execution, and concurrency)	1.8*

A star (*) indicates that the difference is statistically significant ($p<0.01$).

about the tacit knowledge were the most difficult to answer. A possible interpretation could be that the used research method was not suitable or the questions were poorly designed. However, we interpreted that the answering difficulties were mainly due from the topic itself; that is, the topic is genuinely difficult.

It is possible that the respondents do not remember or cannot describe skills that have been automated already several years ago. For example, adults often have difficulties to describe how bicycle is ridden or car is driven. We tried to minimize this problem by dividing the questions in two parts and adding an explanatory text before the questions.

5.2 Conclusions and implications to education

The skills listed can be divided into two main categories: skills associated with composition and skills associated with comprehension. The composition category obviously includes skills that are related to the mastery of the programming languages and environments used. Other important skills associate with having an inherent model of the goal in one's mind, designing interfaces and abstractions, mastering and developing one's own working process, for example. The comprehension category includes skills such as understanding the program as whole, ability to notice isomorfisms with other known problems, ability of change fluently view to the code in various aspects, for example.

On a general level, the results confirm that different comprehension-related tasks are an important part of software developer's cognitive skills. Approximately 40% of the items mentioned by the respondents can be classified as comprehension-related tasks. Obviously, this is not at all surprising result because according to the definition presented in the very beginning of this article, cognitive skills enable human beings to comprehend information.

It is obvious that many of the skills listed above cannot be taught directly on the courses. They are highly related with a long experience gathered when programming solutions to different problems. The challenge for education is to design project assignments where students will face problems, in which the mentioned skills are useful, and how to present guidelines for adopting such skills.

On a more general level, we assume that the deployment of the results of this research might increase the proportion of time used into concept exploration, requirements analysis, and design phases but decrease the proportion of time used into implementation phase. For brevity, we mention only two course examples of such development. The first example would be an advanced course that emphasize comprehension. A possible course title could be "Refactoring." During a refactoring course, a student should repair and/or partly rewrite a program (maybe 2000-3000 lines) that contains different kind of mistakes and bad planning choices. During the task, a student has to read and thus comprehend a program written by others. Moreover, he/she should argue about the findings made, and how the code should be improved.

Second, from the composition viewpoint a possible course title could be "Software design workshop." This course would emphasize analyzing and decision-making skills related to design. The course would contain an open or semi-open design problem that can be solved using several different strategies and tools. The student group should compare various options, argue their pros and cons, and finally evaluate the result.

5.3 Future research

Next are mentioned three possible research settings that we think as interesting for a follow-up research. On purpose, only research settings that use Delphi method are mentioned because our research is a Delphi study.

- Researchers of psychology of programming could be asked as respondents, not experienced software developers. For example, the editors of the book Psychology of Programming might be possible candidates. It would be interesting to compare the results of these two respondent groups. It is possible that researchers of this field can mention some skills that software developers can not—and vice versa. An experienced researcher of psychology of programming might mention, for example, 10–30 cognitive skills when a respondent of this study mentioned only 3–5 skills.

- The respondents could be from an other country than Finland because there might be some cultural differences. We assume that cultural differences related to cognitive skills of software developers are small. However, it would be interesting to explore if this is the case.

- Also a third questionnaire round could be organized. In our research, we stopped after the second questionnaire round because we had promised to the respondents that participating would take 1–3 hours. We did not stop because we though that nothing interesting could be found during a third questionnaire round.

If a similar research will be repeated in the future, we suggest that (a) the division composition versus comprehension, and (b) the definition of cognitive skills that is given in the beginning of Section 1 would be used also in the questionnaires. In addition, we suggest that the first questionnaire would concentrate completely or mainly on cognitive skills. In our research, the questions about cognitive skills were only a small part of the first questionnaire.

6 Acknowledgements

We thank emeritus professor Veijo Meisalo from the University of Helsinki for suggesting use of the Delphi method and PhD Sari Kujala from the Helsinki University of Technology for commenting manuscript of this article.

References

Brooks, R., 1983. Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies 18 (6), 543–554.

Capretz, L., 2003. Personality types in software engineering. International Journal of Human-Computer Studies 58 (2), 207–214.

Conover, W., 1999. Practical nonparametric statistics, 3rd Edition. John Wiley and Sons, New York.

Detienne, F., 2002. Software design—Cognitive aspects. Springer, London.

Engel, G., Roberts, E., 2001. Computing Curricula 2001. Computer Science. Final report, December 15, 2001. IEEE Computer Society and Association for Computing Machinery. Retrieved on October 28, 2004, from the IEEE Computer Society web site: http://www.computer.org/education/cc2001/cc2001.pdf.

ERIC Thesaurus, 2004. ERIC Thesaurus. Retrieved on April 27, 2004, from the Educator's Reference Desk web site: http://www.ericfacility.net/extra/pub/thessearch.cfm.

Greeno, J., Simon, H., 1988. Problem solving and reasoning. In: R. C. Atkinson, R. J. Herrstein, G. Lindzey and R. D. Luce (Eds.): Stevens Handbook of Experimental Psychology, vol. 2.

Kitchenham, B., Pfleeger, S., 2002. Principles of survey research. Part 5: Population and samples. Software Engineering Notes 27 (5), 17–20.

Stanislaw, H., et al., 1994. A note on the quantification of computer programming skill. International Journal of Human-Computer Studies 41 (3), 351–362.

Surakka, S., 2004. Supplementary material for article "Cognitive skills of experienced software developer: Delphi study". URL: http://www.cs.hut.fi/u/ssurakka/papers/Delphi2/index.html.

Wiedenbeck, S., 1985. Novice/expert differences in programming skills. International Journal of Man-Machine Studies 23 (4), 383–390.

Wilhelm, W., 2001. Alchemy of the Oracle: The Delphi technique. The Delta Pi Epsilon Journal 43 (1), 6–26.

Analysis of job advertisements: What technical skills do software developers need?

Sami Surakka
Helsinki University of Technology

sami.surakka@hut.fi

Abstract

An American web recruiting service was used to find the most common technical skills sought in job advertisements for various software developer positions. Data was collected between January and July 2004. The top five skills sought were Windows, Java, C++, SQL, and Unix. Distributed technology skills were analyzed thoroughly because as a consequence of World Wide Web technology, these skills might be required now more often than ten years ago. Results about distributed technologies showed (a) 40% of positions had some skill for the distributed systems mentioned. (b) Microsoft's and Sun's technologies were required approximately as often. (c) These skills were required more often in senior-level than in entry-level positions. This was the only noticeable difference between entry-level and senior-level positions.

1 Introduction

In this study the goal has been to identify technical skills that are important for software developers' work. The research origins from the need to better understand what kind of topics and skills should be included in the Masters level education of software systems specialists in the Helsinki University of Technology. Job advertisements were analyzed to find out the most common technical skills sought in for various software developer positions.

Unlike most previous job advertisement analyses, this research was targeted particularly to technical skills needed in software developer positions. The four main questions were: (1) What skills were needed most in positions for programmers, software developers, and software engineers? In particular, distributed technology skills were analyzed thoroughly because as a consequence of World Wide Web technology, these skills might be required now more often than ten years ago. (2) What were differences between these three job titles? (3) What were the differences between entry-level and senior-level positions? (4) How well do entry-level job requirements match the requirements of a typical bachelors program in computer science?

In the USA, web recruiting services are now dominant in the information technology job advertising market. The biggest service is Dice (http://www.dice.com) that was used for this research because a large number of advertisements was necessary for some parts of this research (e.g., finding large enough sample of entry-level positions).

The results of this study might be useful for training departments of companies, training institutes, and curriculum designers in universities—in particular for those educators who are responsible for Software Systems specializations. Students might use the results when they are selecting elective computer science courses, especially in industry-oriented masters programs. A software developer working in industry might want to compare his or her skills to the results of this article. The results might even reveal something about what technologies were used in new projects if it is assumed that new employees are often hired for such projects. From this viewpoint, the competition between Microsoft's and Sun's technologies for distributed systems is particularly interesting.

The structure of the article is the following. First, the related work is considered in Section 2. In Section 3 the research method is described. The results are presented and analyzed in Section 4. Finally, the research is evaluated, implications to education are considered, and future research is considered.

2 Related work

For any reader wishing to get an overview of research into IT needs assessments, two good starting points are (Nakayama and Sutcliffe, 2000) and (Nakayama and Sutcliffe, 2001). Most previous relevant research has been carried out by educators and researchers who work for information systems/technology (IS/IT) degree programs, not for computer science (CS) or software engineering (SE) programs. 26 publications that are related to this research were found. 22 (85%) of them were from the area of IS/IT. The results have been typically published in publications like the MIS Quarterly and the proceedings of ACM's Special Interest Group for Computer Personnel Research (SIGCPR, currently merged with SIGMIS).

These 26 relevant publications were classified according to the research methods: 9 (35%) were content analyses of job advertisements, 8 (31%) were surveys, 3 (12%) were literature surveys or research in progress reports, 1 (4%) was interview, and 5 (19%) used more than one method.

Next, previous content analyses are described shortly. Main topic of (Adelman, 2000) was certification. In addition, he has written a WWW page (Adelman, no date) that contains results about technical skills such as C++. However, the sample apparently included all kind of IT positions, not just software developer positions. Arnett and Litecky (1994) targeted their research to technical skills for IS positions and found that most wanted skills were PC-LAN, relational databases, Unix, C, and Cobol. (Litecky and Arnett, 2001) and (Prabhakar et al., 1995) are extensions for the previous Arnett and Litecky's research. (Gallivan et al., 2002), (Maier et al., 1998), and (Todd et al., 1995) are trend analyses in IS field. Maier et al. (1998) analyzed individual technical skills such as C, Cobol, and Unix; the samples were from the years 1978–1994. Gallivan et al. (2002) counted proportions of different job titles and averages of skills per advertisement but they did not analyze individual technical skills. Todd et al. (1995) analyzed technical, business, and systems skills for programmers, systems analysts, and managers. Their samples were from the years 1970–1990.

Beside the previous scientific publications and articles in professional magazines, there are two non-scientific reports that are worth mentioning: ITAA report (Information Technology Association of America, 2002) and quarterly reports from a British research company (e.g., Salary Services Ltd. (2004)). The content analysis of job advertisements in the ITAA report (pp. 45–53) is based on Dice's data, and thus, the data source is the same as in this research. The research conducted by Salary Services Ltd. is a content analysis of British job advertisements from several newspapers and web recruiting services. It ranks 150 different skills and reveals which programming languages are required most often, for example. Although (Salary Services Ltd., 2004) is not a scientific publication, in my opinion, it is convincing or even impressive.

From other type of research, the most relevant is the survey by Lethbridge (2000). He asked respondents from about 75 topics: How much they had learned about it in their formal education, how much they knew about it at the time of answering, and how important the topic has been for their career?

The following characteristics are original or different to this research when compared to previous job advertisement analyses: (a) This research is limited to software developer positions and kind of digs deeper in the area of technical skills than previously. (b) Advertisements for more scientific or engineering-oriented positions have been sometimes excluded from the previous samples for an obvious reason: these positions are not very suitable for IS/IT graduates. I work at a university of technology and, therefore, included also more scientific or engineering-oriented positions in the sample. (c) Results about distributed technologies, comparing differences between entry-level and senior-level positions, comparing differences between job titles, and comparing requirements in job advertisements against the degree requirements are new (Section 4.2).

3 Method

Content analysis is a method that is widely used in communications research. Some good properties of content analysis are: (a) it is a non-disturbing method because data occurs regardless of whether the research is carried out or not, and (b) it is often possible to get a representative sample. In this research, a quantitative content analysis of job advertisements was carried out; that is, the frequencies of different phrases such as C++, Java, WebLogic, and "operating systems" were simply calculated.

In February and July 2004, I searched from Dice advertisements that had the job titles Programmer, Software developer, or Software engineer. The searches produced 9680 advertisements. From these 9680 advertisements it was searched for technical skills, using phrases such as Java, SQL, TCP/IP, and Windows. These phrases were typically names or abbreviations of different programming languages, operating systems, database vendors, and protocols. Also some more general phrases like "embedded", "object-oriented", or "relational" were used but this was not common. Note that during this part of analysis the advertisements were not read but only Dice's search function was used. However, a part of the results was calculated with five smaller samples (N=41...334). The advertisements of these smaller samples were read and coded manually.

The following statistical equations, tests, and limits were used. The two-sided confidence intervals for proportions were calculated using equation from (Milton and Arnold, 2003, p. 315) and $\alpha = 0.01$. The Z test for proportions (e.g., ibid., p. 324) was used to test if the difference between two proportions was statistically significant. For statistical significance, the following limits were used: not significant $p \geq 0.05$, almost significant $p < 0.05$, significant $p < 0.01$, and very significant $p < 0.001$.

4 Results

The results are divided into three subsections. First, the most common platform, programming language, and database skills are presented in the subsection "Updating earlier results", where the results of this research are also compared to prior findings. Second, in the subsection "Results characteristic for this research" the results from topics that are more characteristics for the approach used in this research are presented. Based on the literature survey, the topics of the second subsection have been researched only a little or not at all previously. Third, in the subsection "Verification of results" some results from other data sources are presented.

4.1 Updating earlier results

In Table 1, the top five platforms, programming languages, and databases are presented. For example, the proportion of Java was 35% as Java was mentioned in 3359 advertisements and the number of advertisements was 9680. In each column, the sum of the proportions can be greater than 100 because one position could be classified in more categories than one. Below the table are presented the confidence intervals for the worst cases inside each category.

Only the coding principles of the column "Platforms" is explained because they are less clear than the coding principles of the two other columns. The following categories were used: Macintosh, Mainframe/midrange, Unix, Windows, and Cross-platform. For example, Windows refers to those positions where some Windows operating system or Windows based software such as Visual Basic or SQL Server was mentioned. Products that are available for both Windows and Macintosh (e.g., Word and Excel) were classified as Windows if Macintosh was not explicitly mentioned. The category "Cross-platform" refers to positions where only cross-platform products such as Oracle were mentioned.

The results of this research were compared against the results in (Information Technology Association of America, 2002), (Arnett and Litecky, 1994), and (Litecky and Arnett, 2001). For brevity, the detailed results from these publications are not presented but only the most

Table 1: Top five platforms, programming languages and databases.

Rank	Platform	Programming language	Database
1	Windows 42%	Java 35%	Oracle 22%
2	Unix 29%	C++ 34%	SQL Server 11%
3	Mainframe/midrange 23%	C 26%	DB2 7%
4	Cross-platform 17%	Visual Basic 10%	Sybase 5%
5	Macintosh 0%	C# 9%	Access 4%

Confidence interval is ±5% (N=224) for platforms, and ±1% (N=9680) for programming languages and databases (except ±5% and N=224 for Access).

important changes are mentioned. Not very much has changed during the past 5–10 years. For programming languages, the biggest change has been the strong increase of Java. For databases, the biggest change is the increased need of Microsoft SQL Server. The order of platforms is the same as 5–10 years ago when the category "Cross-platform" is excluded from the comparison.

4.2 Results characteristic for this research

Results about distributed technologies, differences between job titles, differences between entry-level and senior-level positions, and comparing requirements in job advertisements against the degree requirements are presented in the following subsections.

4.2.1 Distributed technologies

For distributed technologies three categories were used: Microsoft, Sun, and Other. A position was classified in a certain category if at least one skill of the category was mentioned. It is possible that one position was classified to several categories. The skills of each category are presented in the following lists:

- Microsoft: .NET, Active X, ASP, DCOM, IIS, and MTS
- Sun: EJB, J2EE, JSP, RMI, and Servlets
- Other: technologies that do not belong in the previous two categories (e.g., CORBA, Tuxedo, Tibco, WebLogic, WebSphere, client-server, or applications server).

In 40% of the positions at least one distributed technology skill was required or desired. The proportions of categories were Sun 20%, Microsoft 17%, and Other 8%. These results were calculated with the smaller sample (N=224), manual coding was used, and the confidence interval is ±5%. The difference between Sun and Microsoft is statistically not significant and therefore, Sun's and Microsoft's technologies seem to have had an equally strong position. The most common individual distributed technology skills and their proportions are presented in Table 2.

In addition, 98 job advertisements that were published in Computerworld magazine in the year 1992 were analyzed. The purpose was to analyze if the need for distributed technology skills has increased from the year 1992 to 2004. The sample from Computerworld magazine was coded manually. The year 1992 was chosen because WWW technology was released in 1993. One hundred and twenty-one software developer positions were offered in these 98 job advertisements. From these 121 positions eight were excluded because skills were not mentioned. Only in 4% of positions (N=113) at least one distributed skill was required

Table 2: Most common distributed technology skills.

Skill	Abbreviation	Company	Proportion (%)
.NET	-	Microsoft	19
Active Server Pages	ASP	Microsoft	18
Java 2 Enterprise Edition	J2EE	Sun	13
Java Server Pages	JSP	Sun	8
WebLogic	-	BEA	5
WebSphere	-	IBM	5
Enterprise Java Beans	EJB	Sun	4
Java Servlets	-	Sun	4
Internet Information Server	IIS	Microsoft	3
Common Object Request Broker Architecture	CORBA	-	2
Distributed Component Object Model	DCOM	Microsoft	2
Microsoft Transaction Server	MTS	Microsoft	1

Confidence interval ±1%, N=9680.

or desirable. This difference between Computerworld 1992 (4%) and Dice 2004 (40%) is statistically very significant ($p<0.001$).

It was not possible to compare the results of this research against the earlier researches properly because earlier researches have been so IS/IT oriented or not targeted at distributed technologies. Litecky and Arnett (2001) reported that the proportion of WWW was 6%, Internet 9%, and "Network General" 9% in 1999. However, these results were for all IT positions, not just for developer positions.

4.2.2 Differences between job titles

During this part of research, the purpose was to analyze if low-level programming skills would be more common in software engineering positions. According to (Salary Services Ltd., 2004, pp. 294 and 296), (a) software engineers have in depth knowledge of real time or embedded software and associated hardware, and (b) software engineers are employed mainly by the electronics, computer, aviation, and defense industries. I assume that the job titles "Programmer", "Software developer", and "Software engineer" are often used as synonyms. However, it is reasonable to expect that requirements for software engineers on average would emphasize low-level programming skills more.

Table 3 shows some results for the subsamples of programmers (n=5418), software developers (n=924), and software engineers (n=3338). Only those results that best show the differences about the low-level programming skills are presented. It can be noticed that assembler, C, C++, and the phrase "embedded" were more common for software engineering positions. Some differences between the job titles are statistically significant ($p<0.01$). These pairs are marked with small letters. For example, two letters 'a' in the row "Assembler" mean that the difference between the proportions of programmers and software engineers is statistically significant. Thus, there is some evidence that low-level programming skills are more common in software engineer positions.

No earlier job advertisement analyses were found about the differences between software engineers and other developer positions because in previous researches the compared groups have been different, for example, programmers, systems analysts, and IT managers.

Table 3: Some differences in required or desired skills between job titles.

Skill	Programmer (%)	Software developer (%)	Software engineer (%)
Assembler	1 (a	4	14 (a
C	16 (b,c	30 (b,d	40 (c,d
C++	22 (e,f	48 (e	50 (f
"embedded"	1 (g	5	15 (g

Confidence intervals are ±1...4%, n=924...5418.
Letter pairs indicate that the difference is statistically significant ($p < 0.01$).

4.2.3 Entry-level versus senior-level positions

Two groups were compared against each other: (1) Entry-level positions that had no word "senior" in the job title and the number of required working years was 0–1 (N=41). Often these positions had the word "junior" as part of the job title. This sample was collected mainly in March 2004 from Dice, using phrases like "junior" and "jr." (2) Senior-level positions that had the word "advanced", "lead", "principal", or "senior" in the job title or at least five years work experience was required (N=73). These two samples were coded manually. During this analysis, only the required skills were compared. In addition, individual skills such as Java and SQL were not compared because the sample sizes were so small. Only the following criteria were used: (a) at least one common programming language (C, C++, Cobol, Java, or Visual Basic), (b) at least one common database skill (Access, DB2, "database", Oracle, SQL, SQL Server, or Sybase), and (c) at least one distributed technology skill.

The average number of required skills was somewhat greater for the senior-level group. The average was 3.7 for the entry-level group and 5.2 for the senior-level group. The proportions of the different skill groups were: at least one common programming language 68% (entry-level) and 73% (senior), at least one common database skill 38% (entry-level) and 48% (senior), and at least one distributed technology skill 27% (entry-level) and 59% (senior). The difference between the distributed technology skills was statistically very significant ($p<0.001$). Other differences were statistically not significant.

In addition, the average numbers of software development life cycle phases when the number of required experience varied were calculated. The phases presented in the IEEE standard (Institute of Electrical and Electronics Engineers, 1990, p. 68) were used for analysis. There were no big differences. Even the least experienced developers were typically required to take part in more phases than just implementation. Tutoring younger developers and leading small groups of developers were mentioned often for senior-level positions but obviously not for entry-level positions. The proportions of these duties were not calculated because they are non-technical skills.

Maier et al. (1998) wrote that the proportion of advertisements where experience was required was 34–44% in 1978–1994. Arnett and Litecky (1994) reported "In review of the ads, very few required many years of experience. For the most part, the ads requested experience with words such as 'related', 'some', 'one-year', etc." Experience was required more often in Dice's advertisements than reported in these previous researches. In Dice, the distribution of required years was as follows: no experience 2%, one year 1%, two years 10%, three years 13%, four years 7%, five years or more 26%, and not mentioned 41% (N=224, confidence interval ±5%). Thus, some experience was required in 57% of the positions.

4.2.4 Bachelors programs versus required skills

Next, let us consider how well current curricula in the USA correspond to the job market. McCauley and Manaris (2002) reported that in ABET/CAC accredited bachelor programs the

three most common programming languages that were taught first during the academic year 2001–2002 were Java (49%), C++ (40%), and C (11%). In this respect, the match between curricula and the job market is good because these three languages are exactly the same as the three most common programming languages in the job advertisements. This comparison does not imply that all degree programs should use these three languages. There can be other reasons than the popularity of language in industry to choose the programming language used in education—especially the first one. For example, some institutes might use Scheme as the first language because its syntax is simple.

In addition, McCauley and Manaris reported how often various upper-level courses were required. Related skills are presented in Table 4. The table combines results from two surveys and from this research. The column "Proportion" is based on McCauley and Manaris' survey and refers to the number of times a course was required in accredited programs.

The column "Importance" is based on survey (Lethbridge, 2000) that was targeted to software developers. The means indicate how important the respondents thought that related skills are. The scale was 0–5, and a greater mean indicates greater importance. The data is from the Excel file that can be found from (Lethbridge, no date). If the survey had more than one related item per course title, the means were combined for this article.

In the column "Required in advertisements" is presented my estimation how often the related skills were mentioned in Dice's advertisements. For this analysis, the exact proportions of phrases were calculated but they are not presented because the table would become too complex. For example, for course Database Management Systems phrases "SQL", "database", "relational", and "query" were searched for. The respective proportions were 7–32%. Similarly, related phrases from job advertisements for other courses were searched, too. The text "Hardly ever" refers to proportions 0–1%, "Sometimes" to 2–19%, and "Often" to at least 20%.

Table 4: Most common upper-level courses, their proportions in accredited programs, importance of related skills in Lethbridge's survey and estimation how often related skills were required in job advertisements. See the body text of article for the explanation of the column "Required in advertisements."

Course name	Proportion (%)*	Importance (mean)†	Required in advertisements (estimation)
Operating Systems	96	3.3	Sometimes
Programming Languages	87	2.7	Hardly ever
Software Engineering	76	3.0	Sometimes
Architecture	69	2.7	Sometimes
Analysis of Algorithms	67	2.6	Sometimes
Theory of Computation	49	2.2	Hardly ever
Database Management Systems	31	3.3	Often
Networks	18	3.1	Sometimes
Compiler Construction	16	2.3	Sometimes
Artificial Intelligence	9	1.3	Hardly ever
Human-computer Interaction	4	3.3	Sometimes

*Source (McCauley and Manaris, 2002). †Source (Lethbridge, no date).

This part of research was the most problematic. Here, maybe the main finding is not the results presented in the column "Required in advertisements (estimation)" per se but realizing that this analysis has severe limitations because job advertisements do not contain enough suitable information. Simply put, one can find out from job advertisements that some skills are probably important but solving out if some particular skill or subject is *not important* can be

much more difficult. Analysis seems to work quite well for language and product names such as Java and WebSphere. For other kind of search phrases—that are typically more general terms, selection of phrases can have dramatic impact into results. For example, if I had assumed that the course Programming Languages deepens the understanding of programming principles and is thus relevant for any advertisement that mentions programming or a programming language, my estimation would be "Often" instead of "Hardly ever." However, I have not omitted Table 4 because the results show that topics for at least eight out of 11 courses are required sometimes or often.

4.3 Verification of results

A possible methodological problem with this research is that Dice's data was only from the first quarter of 2004. To verify the results also advertisements that were published in the Computerworld magazine in 2003 were analyzed using manual coding. The sample size was 334 that refers to number of software developer positions. There were some differences between Dice's and the Computerworld's results but the results of this article would be similar if the data source were changed from Dice to Computerworld. Some differences were statistically significant but the order of skills was so similar that the differences had only a little practical relevance. For example, the biggest absolute difference was that the proportion of Java was 35% in Dice and 54% in the Computerworld. This difference is statistically very significant ($p<0.001$) but it has a little practical relevance because both results show that the need for Java was high.

In addition, the results of this research were compared against the results of the recent British analysis (Salary Services Ltd., 2004). The most important difference was that, in the UK, the mainframe/midrange platform was mentioned more often than in the USA. In the UK, the proportion of the mainframe/midrange platform was 4% whereas in the USA it was 19%. As a consequence, the mainframe/midrange related skills Cobol and DB2 were also more common in the USA.

5 Discussion

In this section, the research is evaluated, some implications to education are presented, and future research is considered.

5.1 Evaluation of the research

Maybe the biggest problem with the used sample is a possibility that some well-known companies do not advertise in Dice at all or only a little because interested job seekers search job advertisements directly from the web site of the company. I tried to solve if this is the case for the following companies: HP, IBM, Microsoft, and Sun Microsystems. From these four companies, it seems that Microsoft is announcing in Dice only a very small proportion of open positions but other three companies do announce in Dice. In October 14, 2004, Microsoft offered approximately 930 software development and testing positions in its own web site but only 20 positions in Dice. However, the total number of software developer positions in Dice was approximately 19,000. When compared to these 19,000 positions, the proportion of Microsoft's positions is only approximately 5%. Thus, correcting the sample by including some Microsoft's advertisements from company's own web page would have only a small or moderate effect on results. Obviously, correcting the sample would probably increase proportions of skills that are related to Microsoft's products.

Typical coding problem for job advertisement analyses is, for example, that part of advertisements are so called fishing expedition ads where almost every common skill is mentioned. In some previous researchs, for example (Litecky and Arnett, 2001, p. 7), these advertisements have been excluded. In this research, fishing expedition advertisements were included because

it was not possible to exclude them when Dice's automatic search function was used. This problem will probably result kind of constant background humming for the most common skills, that is, the proportions are a little greater than if fishing expedition advertisements would have been excluded.

5.2 Implications for CS degree programs

Implications of the previous results are considered only for university-level education, not for training institutions or for an individual software developer working in industry. Further, implications are limited only to the typical requirements of accredited computer science programs in the USA because there are no statistics from other countries similar to survey by McCauley and Manaris (2002). The article by Parnas (1999) concerned the differences between CS and SE programs. He wrote "In the SE program, the priority will be usefulness and applicability; for the CS program it is important to give priority to intellectual interest, to future developments in the field, and to teaching the scientific methods that are used in studying computers and software development." Obviously, the results of this research are more relevant to SE programs. However, I write about CS programs and Software Systems specializations because the number of SE programs is so small. As Parnas put it, "Computer Science departments have tried to fill the gap by including so-called 'Systems' or 'Applied Computer Science' courses in their offerings."

Based on the results of this research, the course Database Management Systems should be made compulsory more often. This being said, it would be fair to suggest which one of the other upper-level courses could be changed from compulsory to elective in order to make room for a Database Management Systems course. However, I do not make such suggestion. As explained in Section 4.2.4, this type of analysis is too problematic to find out subjects or skills that are not important.

The results of this study imply that the need for distributed technology skills has increased during the last ten years. However, the results also imply that distributed technologies are not common enough in entry-level positions to make a distributed systems course compulsory in bachelors programs. These skills are common in mid-level and senior-level positions and therefore, the distributed systems course is very suitable as a compulsory course for a specialization Software Systems in, for example, a part-time or industry-oriented masters program.

5.3 Future research

During the Koli Calling conference I was asked if it would be possible to analyze also the depth of technical skills. In my opinion, this is not possible using job advertisements because they do not contain enough suitable information. However, analyzing depth might be possible using another data source. An alternative data source would be databases or other document archives of personnel departments in mid-size and large companies. These archives might contain detailed job descriptions or other suitable data for this kind of analysis.

I have already started a new job advertisement analysis. This is a trend analysis where the data source is the Computerworld magazine and the samples are from every second year from 1990 to 2004. The main purpose of this analysis is to solve (a) if the number of required technical skills has increased and (b) how the need for distributed technology skills has increased during that period.

6 Acknowledgements

I would like to thank professor Lauri Malmi and professor emeritus Veijo Meisalo for commenting on the manuscripts of this article.

References

Adelman, C., 2000. A parallel universe. Change 32 (3), 20–29.

Adelman, C., no date. A parallel universe, expanded. Retrieved on February 3, 2004, from the American Association for Higher Education (AAHE) web site: http://www.aahe.org/change/paralleluniverse.htm.

Arnett, K., Litecky, C., 1994. Career path development for the most wanted skills in the MIS job market. Journal of Systems Management 45 (2), 6–10.

Gallivan, M., et al., 2002. An analysis of the changing demand patterns for information technology professionals. In: Proceedings of the 2002 ACM SIGCPR Conference on Computer Personnel Research (SIGCPR-02), May 14–16, Kristiansand, Norway. pp. 1–13.

Information Technology Association of America, 2002. Bouncing back: Jobs, skills and the continuing demand for IT workers.

Institute of Electrical and Electronics Engineers, 1990. IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990.

Lethbridge, T., 2000. What knowledge is important to a software professional? Computer 33 (5), 44–50.

Lethbridge, T., no date. 1998 education relevance survey results. Retrieved on June 20, 2004, from http://www.site.uottawa.ca/~tcl/edrel/EdrelData1998.xls.

Litecky, C., Arnett, K., 2001. An update on measurement of IT job skills for managers and professionals. In: Proceedings of the Seventh Americas Conference on Information Systems, Boston, MA, August 2001. pp. 1922–1922.

Maier, J., et al., 1998. A longitudinal study of the management information systems (MIS) job market. Journal of Computer Information Systems 39 (1), 37–42.

McCauley, R., Manaris, B., 2002. Comprehensive report on the 2001 survey of departments offering CAC -accredited degree programs. Retrieved on February 11, 2004, from College of Charleston web site: http://stono.cs.cofc.edu/~mccauley/survey/report2001/CompRep2001.pdf.

Milton, J., Arnold, J., 2003. Introduction to probability and statistics, 4th Edition. McGraw-Hill, New York.

Nakayama, M., Sutcliffe, N., 2000. Introduction to research on IT skill issues. In: Proceedings of Americas Conference on Information Systems 2000, August 10–13, Long Beach, CA. pp. 1930–1934.

Nakayama, M., Sutcliffe, N., 2001. IT skills portfolio research in SIGCPR proceedings: Analysis, synthesis and proposals. In: Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research, San Diego, CA. pp. 100–113.

Parnas, D., 1999. Software engineering programs are not computer science programs. IEEE Software 16 (6), 19–30.

Prabhakar, B., et al., 1995. Boom times ahead! Journal of Systems Management 46 (1), 24–28.

Salary Services Ltd., January 2004. ComputerWeekly. Survey of appointments data & trends. Quarterly survey.

Todd, P., et al., 1995. The evolution of IS job skills: A content analysis of IS job. MIS Quarterly 19 (1), 1–27.

Do Students Work Efficiently in a Group? – Problem-Based Learning Groups in Basic Programming Course

Päivi Kinnunen, Lauri Malmi
Helsinki University of Technology
Laboratory of Information Processing Science

pakinnun@cs.hut.fi, lma@cs.hut.fi

Abstract

Problem-based learning (PBL) is an instructional method of group learning that uses true-life cases or problems as a stimulus for learning process. Since 1960's PBL has been used in teaching many subjects in higher education level. While many positive affects have been found, there are some doubts about the effectiveness of the method, too. Some problems that rise from PBL are group dynamic oriented. In our experience one obvious problem is that there are considerable differences in the groups' interaction and this seems to relate to the efficiency of the groups.

This paper focuses on describing the efficiency of PBL groups' interaction. We describe the interaction in PBL groups working in different ways and how students in them experience groups' way of working. Nine tutorless problem based learning group meetings (6 - 7 students in each group) on an introductory programming course were recorded and observed in total 42 occasions. We were able to distinguish which groups interacted efficiently or inefficiently by using observation, interviews and questionnaires. As a baseline for these results we interviewed and sent questionnaires to three tutored PBL groups to gain information concerning tutor's role in a group.

As a result, we describe characters of efficiently and inefficiently working groups and give some examples how students experienced working in a group. Finally, we discuss the possible reasons why students in some groups worked efficiently while others failed in that. We also consider tutors' role in a group, programming as a subject to be learned in a PBL group and some interventions, which might improve inefficiently working groups' situation.

1 Introduction

Our research origins from the experiences of applying Problem-Based Learning (PBL) on an introductory programming course for several years (Nuutila et al., 2000). We observed that students in PBL groups were motivated and gained good learning results. Encouraged by these experiences we launched a new variation of PBL (Kinnunen and Malmi, 2002) where tutors' role was radically reduced and thus less resource was needed. These modifications were launched to investigate how students could cope with reduced tutor resources and still get the benefit of PBL method. If students could cope with this change, we could apply the PBL method to larger scale (400 - 500 student) courses, too.

We observed students' group meetings in several versions of the modified PBL course, and we soon noticed that some PBL groups worked efficiently (that is, the group reached their weekly learning goals, atmosphere was pro-study and group members gained good studying results) and others inefficiently (that is, the group neglected weekly learning goals or made only little effort to reach them, distribution of the work was uneven, many group members dropped out or their studying results were not good). Our research goal was to identify efficiently and inefficiently working tutorless PBL groups and describe their characters. The motivation for this study was to gain knowledge about, which issues impact on how a PBL group works so that we could help more groups to develop an efficient way of working and thus improve their learning outcomes.

The focus of this paper is how tutorless PBL groups work. We approach this subject through analysing groups' interaction during the group meetings and catering for students' opinions about how they experienced the group meetings. We used several methods to collect

the data: Modified Flanders Interaction Analysis System and Bales Interaction Process Analysis (Flanders, 1965; Bales, 1950/1951; Kinnunen and Malmi, 2004) were used to describe and compare the interaction between the groups. Interviews and inquiries were used to gain information about students' experiences.

Computer programming or tutor's role in a group were not in a main focus of this study. However, to see things in a larger perspective, we interviewed and sent questionnaires to tutored PBL groups that had a tutor present at every meeting (we had another in a slightly different way organised introductory programming course where PBL groups had a tutor present at all group meetings (Nuutila et al., 2000)). This way we got some points of comparison of how students experienced tutors' role in a group. These issues are discussed at Sections 5 and 6.

1.1 What is PBL?

Problem-based learning (PBL) is a learning method (Schmidt, 1983) supporting social constructivist (Vygotsky, 1978; Gunstone, 2002) and experiential learning theories (Dewey, 1938/1953; Kolb, 1984). Studying consists of group meetings and self-directed learning. Students are divided into small groups (5 - 8 students) that are given a case or a problem related to some course topic. The case/problem triggers discussion about the subject. Based on this discussion, students brainstorm for a while to make it clear what they already know about the subject and what they still need to learn to better understand the case/problem. After that the students set learning goals for themselves. These steps are done in an opening session, which is followed by self-directed learning. During the self directed learning each student studies to meet the agreed learning goals. Thereafter the group meets again for a closing session where the students discuss about what they have learned. They try to make a synthesis of all knowledge they have and thus try to better understand the case/problem. A tutor is present at group meetings to help students with the learning process. His/her role is, however, rather a facilitator and a domain expert than a teacher[1]

PBL has been widely used in medical sciences ever since 1960's (Norman, 2002). Since then it has been used in many other disciplines, as well, for example, in law and business schools (Huey, 2001) and natural sciences (Williams, 2001). There are many studies of the effectiveness of PBL versus lecture-based teaching (Albanese and Mitchell, 1993; Colliver, 2000; Norman and Scmidt, 1992; Vernon and Blake, 1993). Some reported problems that hinder effective group work are group-dynamic oriented (Woods et al., 1996).

There are various modifications of PBL. See, for example, (Woods et al., 1996) or Finucane et al. (2001). In our modified PBL version groups had a tutor present only at the first two opening sessions and the first closing session. Primarily tutor's role was to help students to get familiar with the PBL method and help them to get started. After that the groups met alone. However, students had an opportunity to meet the tutor once a week for an hour so that they could sort out any problems they had encountered during the group meeting or self-directed learning.

The paper is organised as follows. First, we describe the target of our research. Then we explain the research methods we used. Next, we present the results with examples. In the discussion we analyse some possible reasons for groups working in different ways and in the conclusion we give some concrete advises for overcoming some difficulties that may arise at group work.

2 Goals and Target

Our research targets were groups in courses where the modified PBL method was used. Data was gathered from three courses during 2001 - 2002. In this paper we focus on nine PBL groups, (6 - 7 students each) which were observed in total 42 occasions. Students were study-

[1] A more precise description of the group meeting procedure (7 step method) that we used can be found at URL http://www.cs.hut.fi/Research/COMPSER/ROLEP/PBL.html

ing an introductory computer-programming course (5 study weeks) at Helsinki University of Technology. Attending a PBL group, which came together 8 - 10 times/course, was one compulsory part of this course. Other obligatory parts were programming exercises, a programming project and an exam. Lectures were available but voluntary.

In addition, we interviewed and sent questionnaires to three tutored PBL groups (6-8 students each) on a parallel course to get data concerning students' experiences of tutors' role in a group. These students were studying another introductory programming course and their problem cases were partially the same as on the target course of this research.

3 Method

To describe and analyse how group works is not a simple task and we decided to approach the efficiency of the group work through interaction. We needed a method that would allow us to document the interaction in a group so that the data from different groups could be compared with each other. For that purpose we used an observation method with which we were able to put groups into order according to different interaction qualities. However, this method does not give information how students experience studying in a group or what the learning results are. Therefore we ended up using several methods, which are described in the following sections.

3.1 Observation

To be able to describe the interaction (by interaction we refer only to conversation excluding, for example, facial expressions and body language), we used modified (Kinnunen and Malmi, 2004) Bales Interaction Process Analysis (Bales, 1950/1951) and Flanders Interaction Analysis System (FIAS) (Flanders, 1965). The basic idea of this method is that the interaction is coded into categories and the resulting sequence of numbers is transformed into a matrix. Coding and presenting conversation in a group this way enabled us to describe interaction in different ways. We were, for example, able to see what kinds of addresses were most common in a group or what kind of addresses followed one another (interaction paths). Frequencies of addresses and interaction paths were expressed as indexes. After collecting and coding the data we had one matrix describing the interaction in numerical form per each observation session. Thereafter we were able to compare the differences between groups more easily and reliably, because researcher's own subjective opinion on the group interaction did not affect the results anymore.

Since this observation method has been used to research other context than what we used it for, we had to do some modifications to the method (Kinnunen and Malmi, 2004). We modified the categories (some categories were added and some were slightly modified) and indexes (some new indexes were created). To find out which groups worked efficiently and which inefficiently, we looked more closely at some aspects of interaction. One obvious point of efficient interaction in any group that aims at studying and learning is that the conversation is factual (Postmes et al., 2001). This also means that the conversation does not lapse into irrelevant topics, at least not for a longer time. Previous researchers have also shown that positive encouragement that group members give to each other's fosters groups' interaction. On the other hand, rude addresses reduce contributions (Wheelan and Williams, 2003; Chiu and Khoo, 2003).

On the grounds of this work we observed more closely the most essential parts of interaction concerning efficiency, which were described as following indexes: CCR (Content CRoss: how factual the conversation was), ITC (Irrelevant Talk Circle: the tendency to get stuck on talking on irrelevant subjects), IT (Irrelevant Talk: how much students talk about irrelevant issues compared to relevant issues, in general) and PE (Positive Encouragement).[2]

[2] An example of how conversation is transformed into a matrix and how indexes are calculated at URL: *http://www.cs.hut.fi/Research/COMPSER/ROLEP/interaction_PBL.html*

By sorting the groups into order by these indexes, we were able to identify efficiently and inefficiently working groups. Sorting was done by setting the groups into order by one index at the time. For example, the group that had the lowest IT (irrelevant talk) index got the serial number one (the less there is irrelevant talk during a group meeting the better) and the group that had the second lowest index got the serial number two and so forth. This way each group got a serial number at each index. Finally, all serial numbers from different indexes related to one group were summed up. This process was done with every group. The group that got the lowest sum was regarded as the most efficiently working group.

3.2 Interview and Inquiry

PBL groups were interviewed at the last group meeting session. Questions concerned how the group had worked, how they felt about not having a tutor present all the time, and how they considered more generally PBL as a learning method.

Students were asked to fill in two inquires: one right after the last group meeting session and the other after the whole course had come to an end. Questions concerned student's opinions on the atmosphere in their group and the distribution of the work among the group members. In addition, students were asked other course related issues, which are not considered in this paper.

Answers from interviews and inquiries were analysed at the group level. That is, all the answers from the same group were read at the same time. This way we got the idea how consistently students in one group felt, for example, about how the group had worked.

3.3 Course results

Additionally, we gathered information about PBL students' course grades and how many of them dropped the course.

Finally, we put all the data together. The division between efficient and inefficient groups, which was done based on the observation method, was verified by inquiries, interviews and students' course grades, and course passing per cents. More importantly, by several methods we got deeper insight into what happened in the groups, how the students experienced it, and how it affected the learning results.

4 Results

Based on data that was gathered by the observation method we were able to sort the groups into the following order. The group that has the lowest sum is regarded as the most efficiently working group (see Table 1). We gathered also students' course grades and course passing per cents (see Table 2)

At this course 5 is the best grade and 0 is the lowest. The significance of the difference between course grade averages was calculated with t-test. It was found out that the difference between the first three groups' members' grades and the last three groups' members' grades was significant ($p = 0{,}005$). There is also a trend in course passing per cents so that the three most efficiently working groups had higher course passing per cents than three last efficiently working groups.

The difference between efficiently and inefficiently working groups is clear when looking at the observation and learning results. In the following, we give an example of characters of efficiently and inefficiently working groups, which gives some inside into how groups' efficiency appeared in practice. Examples are based on students' answers during the interviews and in the inquiries. Students' answers confirmed the results of the observation method. The following characters were present in most cases, though there are some exceptions where a character was expressed only in one group.

Table 1: Groups in order by their indexes describing how efficiently the group worked

group	CCR	PE	IT	ITC	sum
L1-02-R4	1	1	2	4	8
L1-01-R5	4	2	3	2	11
L1-01-R1	3	8	1	1	13
L1-01-R3	2	7	4	3	16
Y1-02-R5	5	3	5	5	18
Y1-02-R3	6	4	6	6	22
L1-02-R2	9	5	7	7	28
L1-02-R1	7	6	9	9	31
Y1-02-R1	8	9	8	8	33

Table 2: Course grades, course passing per cent and drop out

group	grade average	passing %	drop out
LI-02-R4	3.8	57	2
L1-01-R5	4.3	67	0
L1-01-R1	4.2	71	1
L1-01-R3	3.4	71	2
Y1-02-R5	4	33	0
Y1-02-R3	3.5	33	2
L1-02-R2	3	50	2
L1-02-R1	2	33	1
Y1-02-R1	3.3	43	5

Efficiently working group

- Members participated the group meetings and made themselves responsible of studying.
- Preparation (that is, working during self directed learning) varied so that all members did not prepare each time as well as the others.
- All members of the group participated to the conversation.
- Atmosphere in the group was open and relaxed. Members felt that it was easy to ask "silly" questions, too, without being picked on.
- Group members felt that their interaction and the way they worked together developed during the course so that interaction became more efficient towards the end of the course.
- Students felt that the group motivated them to study harder.
- When discussing about new concepts and subject matters it was frustrating that nobody in a group knew for sure how the facts are.

Inefficiently working group

- Many members of the group did not participate to the group meetings or dropped out of the course, which reduced others' motivation and made the atmosphere worse.

- In a group, there were students who had a presumption that they can act as free riders and let the others do the work. Only one or two members of the group prepared to the group meeting.
- Atmosphere varied from open to distant and tired. In some groups it was hard to express own ideas.
- Students had difficulties to come to an understanding on how to work.
- Some members had very strong opinions, which they failed to make comprehensive to the other members of the group.
- The meaning of some aspects of PBL 7 step method was unclear to some students.
- When discussing about new concepts and subject matters it was frustrating that nobody in a group knew for sure how the facts are.

For more detail description see (Kinnunen, 2004). The issues concerning attendance, fair distribution of the work and dealing with dominant student at tutored and tutorless PBL groups have also been detected in previous research (Woods et al., 1996).

5 Discussion

Obviously, there is no single reason for inefficiency in a group. For example, based on previous experiences students have presumptions how much they are supposed to work in a group, or people have very different studying preferences and motivations, or there could be personality clashes within the group etc. Next, we take into closer consideration some of the previously mentioned differently working groups' characteristics. We compare some of these characteristics with data we got from tutored PBL groups.

In a group, there were students who had a presumption that they can act as free riders and let the others do the work. Only one or two members of the group prepared to the group meeting.

The presumptions are something one cannot avoid (Bettenhausen and Murnighan, 1991). First, at the university level there is a strong possibility that each group member has been a part of a group before and therefore has a presumption how much he/she has to put effort to the group work. Secondly, according to authors' own experience, at high school level most group works were not so wide-ranging that they would have really required all group members' investment. Consequently, the whole task was, maybe, carried out by only one or two conscientious group members. Previous experience of uneven work distribution may have left students with the idea that they can freeload in future groups, as well.

If there is a tutor present in a group, this problem of uneven work distribution is smaller. Even though a tutor would be just a couple of years older than the student, he/she has the authority over the group. By authority we do not mean that tutor would rule and act authoritatively. However, tutor's presence gives students a motivation to try harder. Students reported, for example, that they prepared for the closing session better because the tutor was there. It was easier to fail to do one's own share if confronted only with peers.

Group members felt that their interaction and the way they worked together developed during the course so that interaction became more efficient towards the end of the course,

or Atmosphere varied from open to distant and tired. In some groups it was hard to express own ideas.

or In some inefficiently working groups, students had difficulties to come to an understanding on how to work.

Each group member has his/her own working/learning preferences. As the group meets for the first time, there might be many different kinds of presumptions and work preferences.

Thus it takes some time until the group has agreed on how they are going to work together. This process has been investigated in several researches earlier. For example, groups' developmental sequence: forming, storming, norming, performing and adjourning were presented already in 1965 by Bruce Tuckman (Tuckman, 1965). Other researchers (e.g. Jonhnson and Johnson (1975/1997); Wheelan et al. (2003)) have presented their own developmental sequences. However, their models are much alike with Tuckman's sequences.

Tuckman, Johnson and Johnson and Wheelan et al. agree that when the group meets for the first time, members try to work out common orientation and means to get there. After that a group might work for a while until it gets to the storming phase where the working methods and authority relationships are questioned. Storming phase can appear in different ways. In some groups it might just appear as longer addresses where members try to get more power to them. At that point members do not listen to each other's very well. Or it might mean more open and obvious discussion with open disagreements and arguments. Either way, at this point the group is not very efficient concerning the task. If the group gets over the storming phase, it has a possibility to develop efficient interaction and work patterns. Later, disagreements can be dealt with in a progressive way so that they do not put risk on groups' unity. However, it is this storming phase that a group has to deal with first and for some groups this might be too hard. Especially, in a situation where there is no tutor present, students' social skills are tested in the storming phase. Tutored groups did not report difficulties concerning work habits. In these groups a tutor could help the groups get over the storming phase quickly and smoothly.

Many members of the group did not participate to the group meetings or dropped out of the course, which reduced others' motivation and made the atmosphere worse.

Student's willingness to be part of a group and commitment to its work is in many respects decided during the first group meetings. The motivation to be a part of a group is based on the balance whether a person gains something valuable, which is more worthy than the negative side of being in the group. (Bayazit and Mannix, 2003; Pescosolido, 2003) For example, a positive side of being a member of the group might be that one gets many peer contacts, which he/she might use as information source or social support. On the contrary, the time that is needed for group meetings and preparation might be considered as a negative side. As long as the student regards positive sides more valuable, he/she is willing to stay at the group. More importantly, estimation concerning the balance of positive and negative sides and efficiency is done during the very first group meetings.

Students reported that if many members of the group did not participate or dropped out, it had a negative impact on their motivation and groups' atmosphere. This might be due to the fact that when the group gets smaller the residual members have to work more than before and they do not have the big groups' support they could and should have had.

Some members had very strong opinions, which they failed to make comprehensive to the other members of the group.

Some group members might be very dominant due to the previous knowledge or personality. In Woods et al. study (Woods et al., 1996) it was also shown that dealing with dominant group members is challenging both in tutorless PBL groups and PBL groups with a tutor. If a tutor is present at a group meeting he/she can help other students to cope with dominating students in a constructive way, thought it might be very challenging for the tutor.

It has been studied that the lack of personal conflicts in a group fosters commitment to the group. On the other hand, disagreements over factual matters do not affect on commitment. (Bayazit and Mannix, 2003) Therefore it is important that students have adequate social skills to work out any social conflicts at an early stage to prevent them growing too big, which would affect members' willingness to commit to the group. In addition, it is not insignificant what kind of personalities a group consists of. The group that meets several times and has tasks to perform has to have a satisfactory social organisation (Borgatta and

Bales, 1953). The PBL group, for example, needs members that are task oriented, as well as members that are socially oriented. In that way the group has an opportunity to develop pro-study atmosphere and at the same time have a good social community. To be part of that kind of group is rewarding intellectually, as well as socially. To ease the group work it is also recommended that there should not be great differences concerning, for example, knowledge level or age within group members since these factors might complicate groups' activity especially during first group meetings (Bayazit and Mannix, 2003).

When discussing about new concepts and subject matters it was frustrating that nobody in a group knew for sure how the facts are.

This concern came up only in tutorless groups. Students felt that there was a risk that they would understand the meaning of concepts wrong. This uncertainty was sometimes frustrating. In contrary, in tutored PBL groups the tutor is, as some students expressed it (Kinnunen, 2004), a sort of safety net, which makes sure that students are not left with false comprehensions. In a PBL group, a tutor is not supposed to teach or lecture but guide the group with well-placed questions when needed. If we think of computer programming as a subject to be learned in a PBL group, we have seen how important it is that students leave the group meeting with correct information. When learning programming one has to learn concepts and their relations well and then be able to apply them to solving practical programming exercises. Computer programming is as much conceptual knowledge as a skill. Misunderstandings at the conceptual level affect directly to the skill level, too. Moreover, since students are solving programming exercises, they face the problems of misconceptions almost immediately. From this point of view, tutor's role as knowledge safety net in a PBL group is very important.

6 Conclusion

In this paper we have described some efficiently and inefficiently working groups' characters and noticed that there are many reasons why groups work in different ways. This variety of working between the groups is greater in these tutorless groups than what we have observed at the normal course where all PBL groups have a tutor present all the time. This observation emphasises the tutor's role in a group. In the previous discussion section we have considered tutor's role as a social and knowledge safety net. Clearly, a tutor is needed in a group not only for his/her knowledge about the subject but also for the social support he/she can provide for the group. As a conclusion concerning tutor's meaning in a group, we could say that a tutor diminishes the differences between the groups. According to our experience tutored PBL groups work efficiently and students are satisfied. Within tutorless PBL group there is a greater variance. Some groups do not have any problems that would affect their learning as some other have great difficulties. These groups with difficulties are the ones that would need a tutor. Unfortunately it is not obvious, how such groups could be identified already in the beginning.

6.1 Some recommendations

As shown in the previous section, there are many variables that affect how the group works. Therefore one cannot give exact reasons why some groups work efficiently and others do not. However, there are steps that teacher can take to help more groups to work efficiently.

- **The very first group meetings are very important; therefore careful planning and preparation are needed.** This means, for example, that PBL cases should be planned very well and students should have a clear idea about the meaning of different steps they go trough in the group meetings. We recommend that students be informed about PBL and what it requires from student well before the course starts so that they

can plan their schedule. A good idea would also be to let them try a rehearsal case before the course begins so that they get familiar with this type of learning.

- It would also be good if **the first tasks students are supposed to complete would very clearly require all members' investment.** On the other hand, the task should not be too hard either. In that way students would see the benefit of being a part of the group. In a programming course, for example, we have observed that students are well capable of conceptual analysis of the program domain, as well as sketching a solution draft very early in the course, when they do not know many programming concepts. Thus **the PBL cases should include problem-solving tasks in the beginning.**

- **The group should make a rule that clearly discourages free riding.** It should also have some tools to meddle immediately if such behaviour is found among its members. Students should be informed beforehand how much studying time is needed per week and also make it clear to them that their level of commitment affects on other group members' level of commitment and motivation.

- **Students should have some knowledge about group dynamics.** This would help, for example, to get over the difficult storming phase by understanding that it is normal to have disagreements and the main point in this phase is how they are solved. **There might be an idea to bind group work / social skills course with some science courses that is using group work.** This linkage would be productive to both courses.

- **Let the tutor be a part of the group at least until the group has got over the storming phase.**

- A tutor in a group must be aware of group dynamics changes that the group is going through. **The tutor needs proper knowledge and training for how to guide a group when needed.**

References

Albanese, M. A. and Mitchell, S., 1993. Problem-based Learning: A Review of Literature on Its Outcomes and Implementation Issues. Academic Medicine 68 (1), 52 - 81.

Bales, R. F., 1950/1951. Interaction Process Analysis. A Method for the Study of Small Groups. Cambridge, MA: Addison–Wesley.

Bayazit, M. and Mannix, E. 2003. Should I Stay or Should I Go? Predicting Team Members Intention to Remain in the Team. Small Group Research 34 (3), 290 - 321.

Bettenhausen, K. and Murnighan, J. K. 1991. The Development of an Intragroup Norm and the Effects of Interpersonal and Structural Challenges. Administrative Science Quarterly 36 (1), 20 - 35.

Borgatta, E. F. and Bales, R. F. 1953. Interaction of Individuals in Reconstituted Groups. Sociometry 16 (4), 302 - 320.

Chiu, M. M. and Khoo, L. 2003. Rudeness and Status Effects during Group Problem Solving. Do they bias evaluations and reduce the likehood of correct solutions? Journal of Educational Psychology 95, 506 - 523.

Colliver, J. A. 2000. Effectiveness of Problem-based Learning Curricula: Research and Theory. Academic Medicine 75 (3), 259 - 266.

Dewey, J. 1938/1953. Experience and Education. New York: The Macmillan Publishing.

Finucane, P., Crotty, M. and Henschke, P. 2001. Clinical problem solving (CPS) as a teaching strategy: A 'dual teacher' approach. Medical Teacher 23 (6), 572 - 575.

Flanders, N. A., 1965. Teacher Influence, Pupil Attitudes, and Achievement. Cooperative Research Monograph No. 12. U.S Department of Health, Education, and Welfare. Office of Education. Washington: U.S Government printing office.

Gunstone, R., 2002. Constructivism and learning research in science education. In: D.C. Phillips. (Ed.) Constructivism in Education. Opinions and Second Opinions on Controversial Issues. Ninety-ninth Yearbook of the National Society for the Study of Education. Part 1. The University of Chicago Press, Chicago, USA.

Huey, D. 2001. The Potential Utility of Problem-based Learning in the Education of Clinical Psychologists and Others. Education for Health 14 (1), 11 - 19.

Jonhnson, D. W and Johnson F. P. 1975/1997. Joining Together. Group Theory and Group Skills. Boston: Allyn and Bacon.

Kinnunen, P., 2004. Interaction and Experiences in the University Student's PBL group. In Finnish. Vuorovaikutus ja kokemukset korkeakouluopiskelijoiden PBL-ryhmässä. Licenciate's thesis. University of Helsinki.

Kinnunen, P. and Malmi, L. 2002. Problem Based Learning in Introductory Programming - Does it Scale up? Proceedings of Second Finnish/Baltic Sea Conference of Computer Science Education, Report. A-2002-7, University of Joensuu, Department of Computer Science, 38 - 42.

Kinnunen, P and Malmi, L., 2004. Some Methodological Viewpoints how to Evaluate Efficiency of Learning in a Small Grou p— a Case Study of Learning Programming. In Laine, A., Lavonen, J., and Meisalo, V. (Eds.) Current research on mathematics and science education. Proceedings of the XXI annual symposium of The Finnish Association of Mathematics and Science Education Research. University of Helsinki, Department of Applied Sciences of Education. Research Report 253.

Kolb, D. A., 1984. Experiental Learning. Experience as The Source of Learning and Development. N.J. Prentice Hall. Engelwood Cliffs. N.J.

Norman, G. 2002. Learning in practice. Research in medical education: three decades of progress. BMJ 324, 1560 - 1562.

Norman, G. R. and Scmidt, H. G. 1992. The Psychological Basis of Problem-based Learning: A Review of the Evidence. Academic Medicine 67 (9), 557 - 565.

Nuutila, E., Törmä S. and Malmi, L. 2000. Ongelmalähtöinen oppiminen ohjelmoinnin perusopetuksessa. In Hein, I. and Lauhia, R. (Eds.) OPE2. Dokumentoitua opetuksen kehittämistä Teknillisessä korkeakoulussa 1999 - 2000. Teknillisen korkeakoulun Opetuksen ja opiskelun tuen julkaisuja 3/2000, s. 92 - 103. Helsinki: Edita.

Pescosolido, A. T. 2003. Group Efficacy and Group Effectiveness. The Effects of Group Performance and Development. Small Group Reasearch 34 (1), 20 - 42.

Postmes, T., Tanis, M. and de Wit, B. 2001. Communication and Commitment in Organizations: A Social Identity Approach. Group Processes and Intergroup Relations 4 (3), 227-246.

Schmidt, H. G., 1983. Problem-based learning: rationale and description. Medical Education 17 (1), 11–16.

Tuckman, B. W. 1965. Developmental Sequence in Small Groups. Psychological Bulletin 63 (6), 384 - 399.

Vernon, T. A. and Blake, R. L. 1993. Does Problem-based Learning Work? A meta-analysis of Evaluative Research. Academic Medicine 68 (7), 550 - 563.

Vygotsky, L. S. 1978. Mind in Society: The Development of Higher Psychological Processes. Cambridge, Mass.: Harvard University Press.

Wheelan, S. A., Davidson, B. and Tilin, F. 2003. Group Development across Time. Reality or Illusion? Small Group Research 34 (2), 223 - 245.

Wheelan, S. A. and Williams, T. 2003. Mapping Dynamic Interaction Patterns in Work Groups. Small Group Research 34 (4), 443 - 467.

Williams, B., 2001. Introductory physics, a problem based model. In: In B.J.Dutch, S.E. Groh, D.E. Allen (Eds.), The Power of Problem Based Learning. A Practical "How To" For Teaching Undergraduate Courses in Any Discipline. Styles, Sterling, Virginia, USA.

Woods, D. R., Hall, F. L., Eyes, C. H., Hrymak, A. N. and Duncan-Hewitt, W. C. 1996. Tutored Versus Tutorless Groups in Problem-Based Learning. American Journal of Pharmaceutical Education 60, 231 - 238.

Explanograms: low overhead multi-media learning resources

Arnold Pears

Arnold.Pears@it.uu.se

Abstract

An *Explanogram* is an animated online presentation of what was written on a sheet of paper, together with a synchronised sound recording of what was discussed at the time. Penstrokes are stored in a timestamped vector representation in the pen as they are written. When writing is completed the user transfers the data to a web server by ticking a send box on the sheet of paper.

Explanograms are an attractive complement to existing educational resources for teaching computer science for several reasons. They are fast to construct, easy to make available and are easily accessed and reviewed by students and staff. In addition they integrate two of the most important media for explaining the types of complex phenomena which occur regularly in computer science, namely writing/drawing and sound. A key advantage of the *Explanogram* in contrast to other methods is that the dynamics of creation are preserved and can be re-experienced by the learner. Combination of two media also enhances the value of the resource by helping to make it effective for both auditory, text and visual learners.

In this paper we describe the *Explanogram* hardware and system software design. A use case which illustrates the potential of *Explanograms* as a teaching and learning aid in computer science is explored to give a feel for the technique.

1 Introduction

An *Explanogram* is an animated presentation generated by writing on a sheet of paper. It presents the content written on the paper in the sequence in which it was originally developed. The system is based on a system developed by Anoto AB (Sweden). Hardware based on the Anoto invention consists of a pen in combination with a special paper pattern. Using these technological components we have developed a system which gives users the ability to save information from the pen (sent via GPRS or an internet connection) into a database. We have also implemented a replay tool (a Java applet) that extracts pen stroke data from the database and presents them in chronological order in a web browser. Users can interact with the content using a playback interface that permits them to select an arbitary position in the explanation, enable a sound recording (if one has been provided) and "fast forward" and "rewind" within the explanation.

Explanograms originated as an approach to capturing multi-media versions of impromptu explanation; thus making them available to a wider audience. The underlying assumption is that difficult areas of the curriculum are often those that prompt students to present themselves during staff "office hours" and ask a question. In the process of answering those questions staff members often reconsider how to present the material, and perhaps adopt a different approach based on the type of difficulty the student appears to be experiencing. This type of explanation might well have been of value to a larger audience, but the logistics of making it available create a barrier to wider dissemination.

Development of our technological solution to this issue is also based on related observations about how teaching and learning takes place. Firstly, and most importantly, it appears that in some areas a static explanation (for example, a drawing and accompanying text such as might appear in a textbook) is often not as effective as the teacher would like. It seems that the order in which one builds up the drawing and develop accompanying textual notes contributes considerably to the overall clarity of the entire explanation.

We observed that office hours questions often focus on aspects of the curriculum that are difficult to present and explain, and for which the dynamics of the presentation are often an integral element of gaining insight. Such enquiries can also result when an explanation

presented in class matches the learning style of some students and not others. In this case one often attempts to explain in a different manner in order to resolve the problem.

Regretably time constraints in lecturing tend to curtail variation in how lecturers expose students to a single concept. Recent research in student learning of computer networking shows that it can be precisely this exposure to variation that is instrumental in facilitating better understanding(Berglund, 2003). We feel that this result is highly likely to generalise to other areas of CS and IT, futher strengthening the case for developing *Explanograms*.

Explanograms help to reduce this teaching dichotomy by providing a resource that lies somewhere between the office explanation (one to one) and the classroom explanation (one to many). They capture chronology of development and several media in a way that facilitates and simplifies presentation of a single concept at varying levels of abstraction, from different technical viewpoints and in a combination of media suitable for a range of common learning styles.

The remainder of this paper is structured as follows. Section 2 discusses related work and the research upon which *Explanograms* draw, in both technical and educational terms. A brief overview of the technologies used to create and replay *Explanograms* is given in section 3. Teaching uses of *Explanograms* are the focus of section 4. Conclusions and further work form the final section.

2 Related Work

Development of *Explanograms* is based on literature from two distinct research areas, animation and handwritten information gathering and storage, and teaching and learning research in computer science. The following sections discuss some related literature in both these areas discussing the "why" and "what" of *Explanograms* in relation to other technical solutions as well as motivating the approch from the point of view of computer science education research.

2.1 Tools

Technical solutions which capture and present diagramatic material are not new. A range of tools and systems are available that produce almost the same outcome as *Explanograms* . However few combine ease of use and chronological information in the manner that is possible with *Explanograms*.

White/Blackboard capture technologies typically produce static images in an image format such as GIF or JPG failing to capture the sequence of creation for the content of individual "pages" (eBeam.com, 2004). Even in the cases where animated representations are produced these are typically stored on a directly connected computer and require additional effort to integrate into a cohesive set of online web resources. In addition synchronised sound recordings are not available.

Perlin's online animation tool "Draw Pad" (Perlin, 2002), is inspired by similar observations to ours, and captures the time element. Unlike an *Explanogram* this system appears not to permit users to vary the speed of replay or provide additional synchronised media (such as sound). The demo version is web based and lacks the natural feel of the *Explanogram* pen and paper interface, though Perlin comments that stylus-based input devices and tablet PC's provide more natural user input interfaces.

Shared whiteboards in online chat programs such as NetMeeting and online course management systems (such as Blackboard and WebCT) also provide some similar functions. However they also lack the user friendly nature of the pen and paper interface, requiring the user to draw using a mouse, and frequently do not preserve the chronology of development, which we feel to be a key aspect of effective explanations.

In addition to the systems discussed above there are a wide range of multi-media authoring tools which allow the user to create animations and presentations that include all the elements that constitute an *Explanogram* , often in a more polished form. Here the time investment in

learning how to use the system is considerable, and the effort associated with incorporating the resulting resources into online teaching materials significant. Given these considerable overheads we consider *Explanograms* to be a low overhead alternative which seems likely to provide similar support for learning.

2.2 Education Research

Development of *Explanograms* is informed by recent research in computer science education and pedagogics. Fundamental to the success of *Explanograms* as a learning and teaching tool is an understanding of their role in an educational setting. Three strands of research seem to be particularly relevant and serve to support the claim that *Explanograms* form a valuable addition to the collection of resources available to teachers and learners of computer science.

A key observation that has emerged from recent research into how students learn computer network protocols (Berglund, 2003) is that good understanding is characterised by the ability to choose a level and type of abstraction appropriate to dealing with the problem at hand. Berglund further argues that there is evidence that this ability to shift between ways of understanding phenomena (such as network protocols) is enhanced when the learning situation is designed to expose students to variation in how they can experience or perceive a single protocol.

Use of simulations and visualisations to deepen insight into difficult topic areas in the CS curriculum has also been the subject of considerable research (Naps et al., 2003). One of the major conclusions has been that active involvement of students is a vital ingredient, if improved learning is to be achieved. In such cases it appears clear that visualisations (properly applied) are a powerful agent for achieving improved learning.

Work on adapting teaching materials to different cognitive styles, and learning styles is common in both educational research (Fleming, 1995) and is becomming of interest in computer science education research (Parkinson and Redmond, 2002). Availability of both visual and textual explanations of a concept as well as auditory material helps a wider range of students encounter learning materials that make it easier for them to comprehend the topic matter.

The preservation of chronology of development and the visual development of material might at first seem suited primarily to visual learners. However, this is not really the case. Associated audio streams provide an alternative. In addition, because *Explanograms* are so easy to create, one can easily envisage creating several which deal with the same concept. This allows one to cover a range of perspectives on a single concept in manners suited to a wide variety of learners and learning styles.

Visualisations and *Explanograms* also have clear similarities, and it seems that interaction is also likely to be important for *Explanograms* success as a learning tool. Learners can interact with *Explanograms* in several ways. A searchable "repository" of *Explanograms* allows students to interact with the collection of available resources and choose what seems most useful to them. Each individual *Explanogram* also provides a level of interaction through a user interface which allows the user to drag a progress bar through the chronological sequence, enable and disable sound, review the explanogram at different replay rates, and pause and restart replay as desired.

In a sense *Explanograms* "kill several birds with one stone". Creating an online multi-media explanation becomes as easy as writing on a piece of paper and ticking a send box. Creation of multiple *Explanograms* in response to student questions rapidly builds up the type of variation advocated by Berglund. An additional side-effect of the multi-media nature of *Explanograms* is that they tend to cater to a wider range of learning styles.

3 Explanograms

Explanograms are based on a commercially produced pen which is capable of sensing and recording pen strokes made on a normal sheet of paper upon which a barely visible background pattern has been printed. The pattern on the paper provides coordinate information in a very large pattern space, and the pen associates a time stamp and coordinates associated with the path of the pen on the pattern space between each "start" and "stop" action of the pen. A "start" action occurs when the pen is pressed onto the paper (whereupon timestamps and coordinates start to be recorded) and the "stop" action occurs as the pen leaves the paper (terminating coordinate/timestamp recording) for the current "pen stroke". Pen location is sampled at 50 Hz.

Automated storage and presentation of explanograms is possible since the pen automatically uploads the penstrokes to a designated computer where we then store them in a database. This database of timestamp and penstroke information is then processed and indexed with the other media forms available in order to present explanogram visualizations with associated audio and visual media to the user.

3.1 The Anoto Technology

3.1.1 The Paper

The paper pattern consists of a fine mesh of dots displaced from the verticies of a grid. The digital pen detects this pattern using infra-red illumination and a camera and requires only a small portion of the grid (6x6 dots) to locate its position in the total pattern space. The combinations of unique dot displacements from the grid define an area exceeding 60 million km^2 in which each 6x6 dot area is unique. This corresponds to a writing surface area equivalent to the land area of Asia and Europe combined.

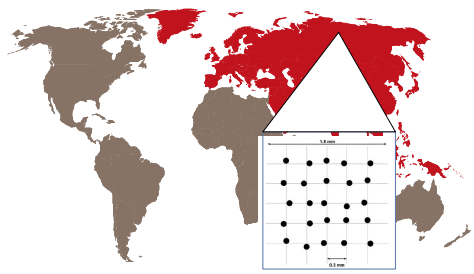


Figure 1: The pattern grid and its coverage. Every square 2 x 2 mm is unique

The camera in the digital pen operates in the infrared spectrum and the pattern has to be printed with carbon-ink. Due to the absorption of infrared light by the carbon ink the pattern is visible to the pen, even through drawn or printed material created using non carbon ink. Thus, when drawing with the digital pen the camera only sees the pattern on the paper and not the written ink or other material printed in non-carbon ink.

Some special pattern areas are defined as commands to the pen. These patterns are called "Pidgets" in the Anoto terminology. Pidgets are interpreted in the digital pen and are used to execute common functions that change the pen status. That is, set the virtual ink colour and line thickness that the pen reports to applications, as well as implementing interface commands (such as "send data"). When the user places the pen on a pidget the digital pen executes an internal function and reports success or failure to the user through vibration feedback.

More detailed information about the Anoto technology, pre-preprogrammed template functionality and the paper pattern is available from the Anoto website (Anoto, 2004).

3.1.2 The Pen

Anoto's Pen Technology, in combination with the paper pattern descibed above, permits the generation and recording of highly accurate pen stroke data by writing on ordinary paper upon which the Anoto pattern has been printed. Several companies have licensed the basic technologies and produce digital pens (Nokia, 2003; Ericsson, 2002; Logitech, 2002) which allow a user to store and easily transmit what is written on a sheet of paper. The digital pen contains a camera, embedded processor, Bluetooth radio transceiver and an ink cartridge.

Commands to the digital pen are all implemented through interactions with the paper. There is limited feedback display on the digital pen, only light diodes and a vibration device which indicate operational modes and provide feedback to the user. Since the digital pen can communicate using the Bluetooth transceiver, it can also give feedback to the user by displaying messages through pen interaction applications running on mobile phones and computers, however this ability is very limited in Nokia and Logitech models.

The Sony Ericsson pen is no longer on the market and has the limitation that it is only possible to use it in conjunction with model T39 and T68 mobile telephones. Use of this pen is not possible with computers and bluetooth base-stations.

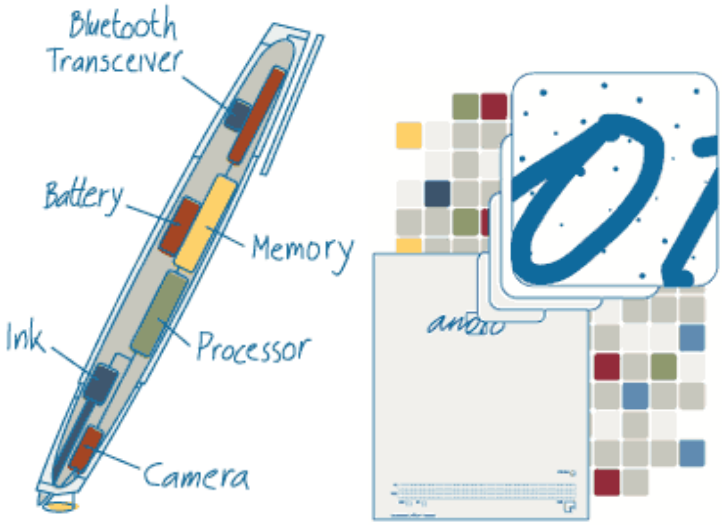


Figure 2: The Anoto Pen and the Anoto Pattern

3.2 Explanogram Service Architecture

Flexible, location independent, generation of *Explanograms* depends on a more general service infrastructure. This is provided by a combination of services hosted at Anoto AB in Stockholm, Sweden. Data from internet connected pens is automatically forwarded to a Paper Lookup Server (PLS) where the owner of the paper is determined together with a final destination server for the user data that has been sent from the pen.

A Java servlet on the server at the ultimate destination receives the data sent from the pen and decodes it using an application programming interface (API) provided in the Anoto "Application Development Kit" (ADK). Once the data has been decoded we store penstroke and timestamp information in a database. This data is then immediately available for replay using the Java replay applet.

In our case data is sent to the *Explanogram* server from bluetooth equipped pens via either a PC with an Internet connection, or a mobile phone with a GSM/GPRS connection. Data transfer from USB pens is also supported in conjunction with an Internet connected computer. An overview of the server lookup and data transfer process is shown in Figure 3.

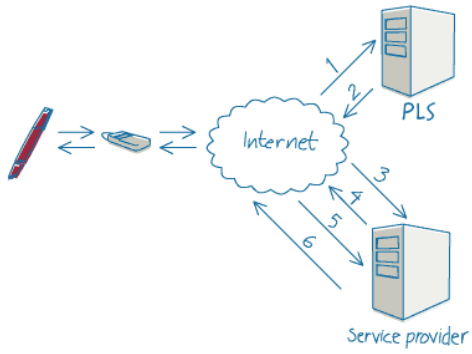


Figure 3: Internet Functionality: Using the pen and paper in a mobile setting

The ability to use the pen in mobile configuration means that explanograms can be created anywere. Thus one can imagine creating an explanogram during a teaching break in a classroom, or recording a text only explanogram at a hotel while at a conference in response to an email question.

The advantage of the USB and PC configuration is that it simplifies synchronised recording of sound which can then be easily associated with the explanogram. Integrated sound recording is not currently available when using the pen and a mobile telephone unless the mobile phone supports audio note taking. This is achievable in with combination PDA's and telephones such as the recent Palms and the Sony-Ericsson P800 and P900 series telephones.

Certainly an email reply to a question is also possible, but creating electronic pictures and diagrams is time consuming. The explanogram captures the speed of handwritten explanations and makes it rapidly available electronically, thus significantly increasing the effectiveness of the teacher.

4 Uses for *Explanograms*

To date *Explanograms* have been used in two courses in computer networking, primarily with distance students. During these trials user experiences were surveyed using questionaires. In general feedback was positive, though this may well be due to the Hawthorne effect Mayo (1939). Eight students participated in the study and reviewed sample *Explanograms* on the course web site. All students reported a positive experience with the system, but noted that a better user interface was needed where speed of replay could be controlled. Some students also mentioned that they felt that audio soundtracks would be useful.

We interpreted the generally positive feedback as motivation to continue with development of the *Explanogram* service. A new interface was developed during 2003 which provides replay controls and integrated sound recordings in response to feedback on the initial prototype. Futher studies of student experience with *Explanograms* and their adoption of them as a learning resource are currently underway based on the new replay tool.

To give an impression of how *Explanograms* have been used in teaching situations we present a sample *Explanogram* scenario.

The "slow start" algorithm is a probing based sending rate adjustment mechanism used to control the data window size when sending data over a TCP connection. TCP is one of the fundamental protocols used in the Internet to reliably transfer data between applications. Competition for network transmission bandwidth means that a mechanism is needed to control how much of the available bandwidth is consumed by each individual TCP connection.

An example scenario is typically used in lectures to explain the operation of this mechanism. This type of explanation is eminently suited to the *Explanogram* approach, since undestanding how the mechanism operates depends on understanding how the behaviour diagram develops over time. To give the reader a feel for what this looks like we have extracted

some snapshots from this *Explanogram* in Figure 4. The images on the left hand side of the figure are from the original flash animations, while those on the right show similar content in the new Java Applet based replay tool.

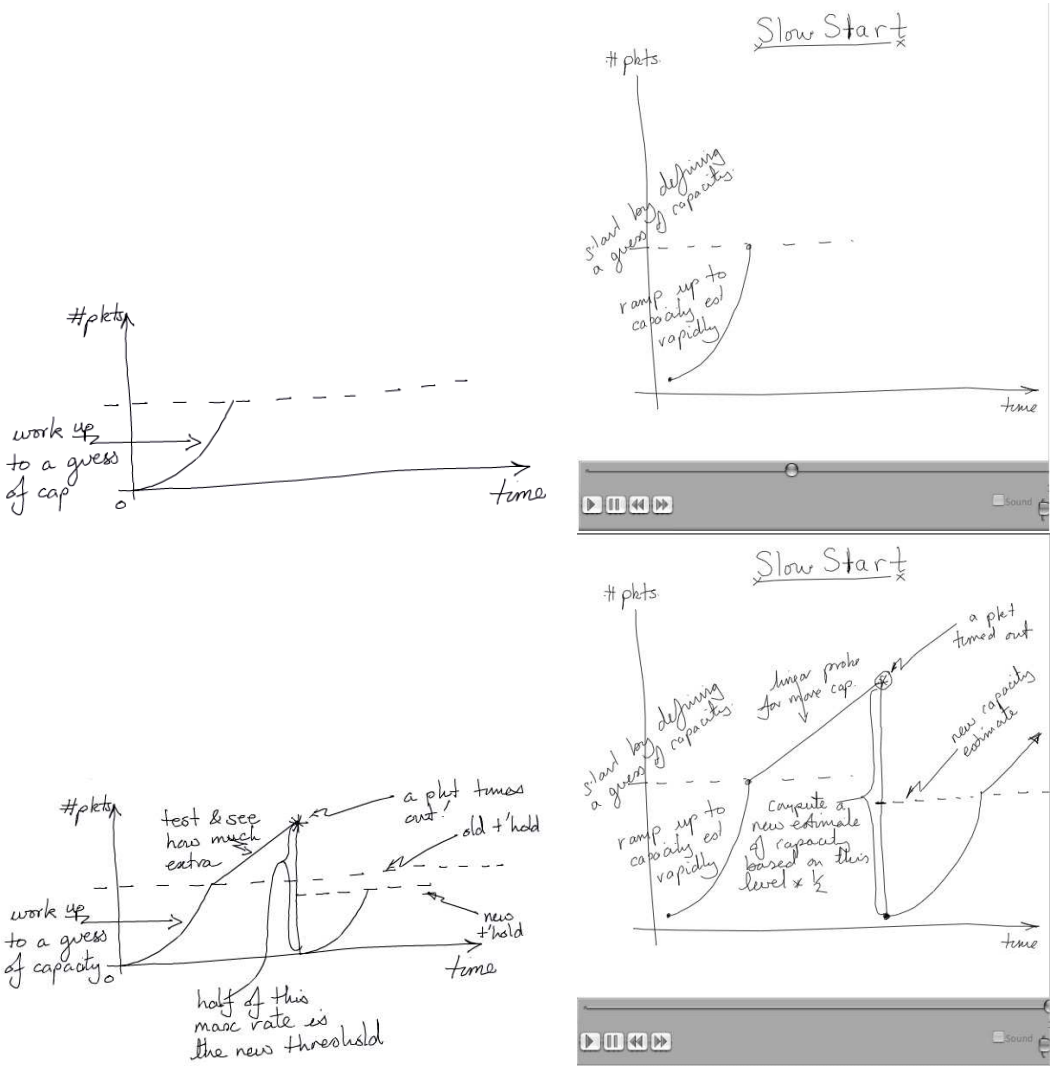


Figure 4: Slow Start Operation

5 Conclusion

This paper discusses the technologies and software systems surrounding a new approach to creating online multi-media resources called *Explanograms* (http://explanogram.it.uu.se). In comparison to many other techniques for capturing such information *Explanograms* offer greater richness (by incorporating a wider range of media) and preserving chronology of creation. However the major advantage of the technique is its simplicity and low overhead approach to creating complex resources. A flexible user interface allows online interaction with an *Explanogram* repository, where material can be selected and reviewed using an intuitive playback interface design.

We believe that *Explanograms* constitute a powerful learning resource for several reasons. They integrate a range of media, making them more approachable and useful to a wider range of learners with differing learning styles (text, visual, auditory). Because they are so easy to create and index it is easy to create several different *Explanograms* about a single problem

topic enhancing the variation in ways of experiencing and learning about that topic. Finally, preserving the chronology of development helps to make an *Explanogram* clearer and more engaging than the printed page.

Future work on evaluating the impact of using *Explanograms* in teaching computer science courses in computer networking and realtime systems is underway. Future system enhancements include refining the repository search facilities and implementing support for new input devices.

References

Anoto, 2004. Anoto website. Http://www.anoto.com.
URL `http://www.anoto.com`

Berglund, A., 5–8 November 2003. What is good teaching of computer networks? In: Seidman, S. (Ed.), Frontiers in Education (FIE'03). Vol. 3. pp. S2D–13–18.
URL `http://user.it.uu.se/ andersb/fo/FIE-Berglund.pdf`

eBeam.com, 2004. eBeam electronic whiteboard system. `http://www.ebeam.com/`.

Ericsson, 2002. Ericsson chatpen. `http://www.ericsson.com/`.

Fleming, N., 1995. I'm different; not dumb. modes of presentation (vark) in the tertiary classroom. In: Zelmer, A. (Ed.), Proceedings of the 1995 Annual Conference of the Higher Education and Research Development Society of Australasia (HERDSA). Vol. 18. pp. 308–313.

Logitech, 2002. Logitech io pen, usb digital pen. `http://www.logitech.com/`.

Mayo, E., 1939. The human problems of an industrial civilisation. New York, MacMillan, additional details from http://www.psy.gla.ac.uk/ steve/hawth.html.

Naps, T., Rodger, S., Velazquez-Iturbide, J. A., Rössling, G., Almstrum, V., Dann, W., Fleisher, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., 2003. Exploring the role of visualization and engagement in computer science education. ACM SIGCSE Bulletin 35 (2), 131–152.

Nokia, 2003. Nokia bluetooth digital pen. `http://www.nokia.com/`.

Parkinson, A., Redmond, J. A., 2002. Do cognitive styles affect learning performance in different computer media? ACM SIGCSE Bulletin 34 (3), 39–43.

Perlin, K., 2002. "draw pad: The elusive 'animated napkin sketch'". `http://mrl.nyu.edu/ perlin/draw/`.

Point-and-Click Logic

Matti Nykänen
Department of Computer Science
University of Helsinki
FINLAND

matti.nykanen@cs.helsinki.fi

Abstract

Students of proof theory, a branch of formal logic, can benefit from computerized tools. We describe the principles behind one such tool called PROED. This tool is targeted especially to novice students, and therefore it is designed to support effortless exploratory use. We moreover argue that focusing on root-first proof construction in Sequent proof systems helps attain this effortlessness.

1 Introduction

Kapur (2004) recently wrote: "During the past ten years, theorem proving has been playing an increasingly important role in education. One reason is the improvement in the friendliness of theorem provers; many are now suitable for the nonspecialist [...] With computer-based educational tools profoundly changing the way instruction is imparted, automated reasoning tools are likely to have considerable impact in education and learning in the future." Such automated reasoning tools are basically computer implementations of different *proof theories*, or even meta-frameworks extensible with new proof theories. A proof theory for a logic in turn is a formal system for constructing correct proofs for statements written in this logic, such as those presented by Negri and von Plato (2001). These proofs are constructed step by step according to specific *inference rules*.

When the student encounters his first proof theory, he faces the following problem: He is given the collection of inference rules of the proof theory in question, and he should start composing proofs of logical formulæ using them. However, merely knowing the rules themselves is not enough to know how they can be made to work in unison as parts of a complete formal proof. Instead, the student must experiment with proof construction in order to see the rules in action.

Constructing a formal proof involves not only the reasoning steps themselves but also tedious bookkeeping to ensure that these steps really do connect to each other in an appropriate manner. Such bookkeeping is best left to the computer; otherwise the student's experimentation is hampered by mere notation, which distracts him from the actual subject matter.

This is why for instance the textbook by Negri and von Plato (2001) also provides the PESCA tool, a computer implementation of its proof theories. PESCA has been used in the proof theory courses given by Negri at Helsinki University Computer Science Department.

PESCA is a vast improvement over pencil and paper with respect to bookkeeping. On the other hand, its purely textual interface does not lend itself well to experiments like "What if I applied *this rule* to *that position* of my ongoing proof?": the student must first translate his experiment into a PESCA command which explicitly spells out not only the rule to apply but also the path to the desired position. Worse, if another rule has already been applied at that position, then the student must first *undo* it before the new rule can be applied in its place. Furthermore, once the student gains more confidence, his experiment can ask for a whole *series* of rule applications to be performed starting at the indicated position. In PESCA, the student must spell out the series of corresponding commands explicitly.

Our aim is to explain how the student can perform such an experiment with a *single click or drag-and-drop operation with the mouse* in a graphical user interface (GUI). We shall see how the student's operation suffices to indicate the *last* position and rule to apply. For

certain kinds of proof theories, the computer can then generate the intermediate stages of the series without any further guidance from the user. This leads to a tool where the student uses mouse operations to indicate only the key choices in constructing the proof, while the computer performs the bookkeeping involved in the concomitant series of rule applications. We feel that such a tool supports effortless experimentation with different proof construction strategies.

Our presentation runs as follows. Next, Section 2 explains what kinds of proof theories suit our approach, followed by Section 3 which explains how mouse operations are translated into such proof theories. Then Section 4 reports on the prototype implementation of our design. Finally, Section 5 concludes the presentation.

2 Requirements for the Logics

Let us now examine some main varieties of proof theories with respect to their suitability for student exploration in the manner described in Section 1.

Resolution (Russell and Norvig, 2003, Chapter 9.5), taught in Artificial Intelligence (AI) courses for its easy programmability, is unsuitable: It requires a preprocessing step which *converts the input formula* into a certain normal form on which subsequent proof search operates. This conversion makes it harder for the student to control the search, since the resulting form differs widely from the original formula he set out to prove.

Hilbert (or axiomatic) systems (Restall, 2000, Chapter 4), taught in Mathematical Logic courses for their amenability to metamathematical analysis, do retain the original form of the formula to prove. However, they are unsuitable for another reason: The *axioms*, or the "obvious" starting points of proofs which require no further proof of their own, are often very complicated formulæ themselves. The student is thus expected to master the use of these complicated formulæ right from the beginning of his study.

Hence we seek proof theories which both retain the original formulæ and have simple axioms. Two related candidates are **Natural deduction** (Negri and von Plato, 2001, Chapter 1.2) and **Sequent (or Gentzen) systems** (Negri and von Plato, 2001, Chapter 1.3). Natural deduction is taught in philosophical logic courses, because its inference rules correspond to intuition about how reasoning proceeds: An *introduction* rule brings a logical connective into the proof, while the corresponding *elimination* rule gets rid of it. For example, the introduction rule $\supset I$ for implication reads "if assuming A enabled you to prove B, then you have proved $A \supset B$".

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B} \supset I$$

The corresponding elimination rule

$$\frac{A \supset B \quad A \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C} \supset E \tag{1}$$

is engineered so that introduction immediately followed by elimination cancel each other out to yield the direct proof without the implication. Natural deduction is chosen by for example the ETPS system (Andrews et al., 2004) with its emphasis in educational use.

As natural as this approach is, it too suffers from a drawback: The student is initially faced with only the conclusion C to prove. Suppose he wanted to experiment with rule (1). Where would the new assumption $A \supset B$ come from? While a seasoned logician may well somehow see it from C, the novice student may not. Coming up with just the right assumption is certainly a valuable skill to master in constructing proofs; but here we are designing a tool for learning proof construction to begin with.

A third related candidate is using **analytic tableaux** as in for example the STRATUM system by Janhunen et al. (2004). This method is akin to solving systems of equations with the form "formula A is true / false" by repeated decomposition. However, the idea of Natural deduction and Sequent systems extends in a natural way to modelling other properties of sentences than their truth- or falsehood; an example are the Lambek calculi from formal linguistics (Carpenter, 1997, Chapters 5.1 and 5.2).

We therefore opt for sequent systems. Natural deduction drew conclusions like "formula C is true" below the horizontal line of the inference rule. Sequent calculi instead draw conclusions like "there is a proof of C from assumptions Γ"; these are denoted as $\Gamma \Rightarrow C$. Thus sequent calculi provide an explicit notation for handling assumptions, and this notation is local to the conclusion being drawn. On the other hand, this notation entails a lot of redundant bookkeeping, which makes sequent calculi tedious for experimentation on pencil and paper. However, we can relegate this routine bookkeeping to the computerized tool.

Consider as our example the proof system depicted as Table 1. Introduction rules become right rules operating on a formula on the right side of a sequent, while elimination rules become the corresponding left rules insted. Thus the left rule $L\supset$ corresponds to elimination rule $\supset E$ above. Comparing these two rules shows the effect: the student is faced with a sequent $\Delta \Rightarrow C$ to prove. However, now his choices are explicitly listed within it: he can manipulate either its right side C with the applicable right rule(s) or any member of its left side Δ with the applicable left rule. Moreover, examining the rules in Table 1 reveals that every formula above the horizontal line is present already below the line, either as a part A or B of the formula operated on by the rule or as a member of Γ, the other assumptions that are retained as is by the rule. This *subformula property* remedies the drawback of natural deduction noted above.

Table 1: The sequent system **G3ip**.

The axioms are

$$P, \Gamma \Rightarrow P \tag{2}$$

where the formula P is *atomic*, or contains none of the connectives '&' (conjunction), '$\vee$' (disjunction), '$\supset$' (implication) or '$\bot$' (falsity). Each sequent is of the form $\Delta \Rightarrow D$, where its *right side* D is a formula, and its *left side* Δ is a multiset of formulæ. That is, Δ may contain multiple distinct copies of the same element. The shorthand A, Γ denotes such a multiset from which one such copy of its element A has been singled out while Γ consists of the other elements of the multiset. The inference rules are as follows:

	left rule	right rule(s)
&	$\dfrac{A, B, \Gamma \Rightarrow C}{A\&B, \Gamma \Rightarrow C}\ L\&$	$\dfrac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A\&B}\ R\&$
$\vee$	$\dfrac{A, \Gamma \Rightarrow C \quad B, \Gamma \Rightarrow C}{A \vee B, \Gamma \Rightarrow C}\ L\vee$	$\dfrac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B}\ R\vee_1 \quad \dfrac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B}\ R\vee_2$
$\supset$	$\dfrac{A \supset B, \Gamma \Rightarrow A \quad B, \Gamma \Rightarrow C}{A \supset B, \Gamma \Rightarrow C}\ L\supset$	$\dfrac{A, \Gamma \Rightarrow B}{\Gamma \Rightarrow A \supset B}\ R\supset$
$\bot$	$\dfrac{}{\bot, \Gamma \Rightarrow C}\ L\bot$	no such rule

This suggests a particular proof construction mode: the student starts with the sequent $\Delta \Rightarrow D$ to prove. He selects from it a particular formula, either D or some member of Δ. The computerized tool can in turn extract all the information needed for displaying the new sequent(s) that should appear immediately above the starting sequent $\Delta \Rightarrow D$ (separated by the horizon-

tal line). Then the user can continue with these new sequent(s) in the same manner, until he encounters axioms. Note that the axioms (2) are now easy to recognize, as they only require that the same formula P of the simplest possible kind appears on both sides of the sequent; this in turn remedies the defect of Hilbert systems noted above. This mode of proof construction is called *root-first:* The proof is in fact a tree of sequents, and this mode grows the tree from the ground up. (It is also called *top-down* in accordance with the computer scientists' habit of drawing their trees in an "antipodean" way with the root at the top.)

Although every possible proof can be found with this root-first proof construction mode, the student may well plan his reasoning steps in some other mode as well. A well-known example is the *Modus Ponens* rule

$$\frac{A \supset B \quad A}{B}\ MP$$

which lies at the heart of Natural deduction systems and is a special case $C = B$ of our example elimination rule (1) (Negri and von Plato, 2001, Chapter 1.2). The natural mode to use this rule is *forward* reasoning: from "A implies B" and "A" infer also "B". On the other hand, the corresponding step in our root-first mode is

$$\frac{A \supset B, A, \Delta \Rightarrow A \quad B, A, \Delta \Rightarrow B}{A \supset B, A, \Delta \Rightarrow B}\ L\supset$$

where the reasoning proceeds *backwards* instead: the formula $A \supset B$ is first analyzed into its constituents A and B, which are then analyzed further into axioms. Systems like ETPS (Andrews et al., 2004) and JAPE (Bornat and Sufrin, 1997) allow the student to proceed in either mode, and even mix these modes within the same proof. However, this complicates the student's interaction with the tool: he must indicate both the formulæ to operate on and the mode of operation. This in turn means that he must be aware of these two modes as well. We believe that a single-mode tool is a more appropriate one to begin with.

Incidentally, this distinction between forward and backward chaining of steps arises also in automated reasoning. There backward chaining is found to be the more goal-oriented of the two (Russell and Norvig, 2003, Chapters 9.3 and 9.4). This finding accords with our discussion above: backward reasoning proceeds by splitting more complicated formulæ into their simpler constituents, and is therefore always oriented towards the simple axioms. In contrast, a forward reasoning step does not guarantee progress by itself, but only together with other choices by the student, as shown by our discussion on rule (1).

The student may naturally also encounter a dead end: a non-axiom sequent where no rule applies. Hence the student must also be allowed to *undo* his previous choices. In the tree metaphor, this amounts to pruning away some part of the tree, so that a new branch can be grown into its place using a different combination of rules.

Next, Section 3 explains how all this information can be extracted from the student's action. However, we must first point out a technicality: the rules in Table 1 have the additional property that the unmodified part Γ is passed from the sequent below the line to every sequent above it. That is, the context Γ of the rules is *shared* among the sequents. Another way would be to somehow partition the context Γ among these sequents. However, such partition information would be difficult to extract without additional information from the student.

3 Proof by Pointing

The student's mouse click must be translated into sufficient information about the proof steps that should be taken. This can be done with the *proof-by-pointing* approach suggested by Bertot and Théry (1998, Section 4) as a user interface concept for interactive theorem provers. Let us illustrate this approach with an example.

Suppose that the student wants to prove the transitivity of implication:

$$\Rightarrow ((A \underline{\supset} B)\&(B \supset C)) \supset (A \supset C). \tag{3}$$

Suppose further that he clicks the underlined implication '$\underline{\supset}$'. The natural interpretation is that he wants to apply some inference rule to it, because the user is pointing to it with the mouse as if saying, "I want to use *that thing* in my proof next!". However, that thing to use next is nested inside other connectives:

$$\Rightarrow \boxed{(\boxed{(A \underline{\supset} B)\&}(B \supset C)) \supset}(A \supset C).$$

Since the rules apply to the outermost connective only, we must first peel off the outer boxes one by one. The main connective of the outermost box is an implication '$\supset$' on the right side of the '$\Rightarrow$' (the left side being empty). Hence the rule to use must be $R\supset$:

$$\frac{\boxed{(A \underline{\supset} B)\&}(B \supset C) \Rightarrow A \supset C}{\Rightarrow ((A \supset B)\&(B \supset C)) \supset (A \supset C)}\ R\supset$$

This leaves the next box to peel off. Its main connective is the disjunction '&' which has now jumped over the '$\Rightarrow$' on its left side. This jumping behaviour shows why these boxes must be peeled off from the outside in. The corresponding rule is therefore $L\&$. Now the underlined implication '$\underline{\supset}$' is finally exposed, and it can be peeled off as well with the rule $L\supset$, leaving the following incomplete proof:

$$\dfrac{\dfrac{\dfrac{A \supset B, B \supset C \Rightarrow A \quad B, B \supset C \Rightarrow A \supset C}{A \supset B, B \supset C \Rightarrow A \supset C}\ L\supset}{(A \supset B)\&(B \supset C) \Rightarrow A \supset C}\ L\&}{\Rightarrow ((A \supset B)\&(B \supset C)) \supset (A \supset C)}\ R\supset \tag{4}$$

At this point we stop and wait for the next mouse click from the student, because we have used up all the information from the previous one.

Note that this box nesting is uniquely determined by the structure of the original sequent (3) and the location of the mouse click inside it. That is, we can compute independently what the student's "use that" mouse click means without any additional information from him. The crucial requirement for this independent computation is noting that *the rule for peeling off a box is uniquely determined* by its side in the sequent and connective. We opted for sequent systems in Section 2 precisely because they permit such independent computation in the manner described above. This uniqueness also means that the student is still in full control of constructing the proof: the tool merely calculates the effects of his chosen rule applications, it does not choose any rules on its own. Even so, the student can perform a whole series of rule applications with a single click, as demonstrated. Or if the novice student finds this confusing, he can still work step by step instead; in our example above, he then begins by clicking the conjunction '&' in sequent (3) instead. Thus this approach serves the student not only at the very beginning but also when he has become confident enough to try several inference steps at once.

In our example above, the final action of the independent computation was to apply the rule to the connective that the student clicked. However, this final rule application is not necessarily unique. Such an ambiguous case is clicking a disjunction on the right side of the sequent; this click can mean either rule $R\vee_1$ or $R\vee_2$ of Table 1. In such a case, the computation proceeds until the clicked connective is no longer enclosed in any box, and stops there to wait whether the student chooses subformula A or B. This choice then disambiguates between the competing rules. This disambiguation is naturally automatic for all the rule applications preceding the last one.

Admittedly one can envison proof systems with multiple different rules to peel off a box: for instance, the rule might depend on a combination of several connectives instead of just one as above. However, recalling our discussion in Section 2, this would somehow mean that these different combinations possess different introduction or elimination rules in the corresponding

Natural deduction system. This would in turn mean that the intended meanings of these seemingly separate connectives would in some sense be intertwined after all. We feel that studying proof theory should start with other than such delicate systems.

The student can subsequently continue his proof (4) by clicking similarly inside either of its uppermost sequents $A \supset B, B \supset C \Rightarrow A$ or $B, B \supset C \Rightarrow A \supset C$. However, he soon finds out that the former leads into an infinite regress whereby rule $L\supset$ is repeatedly applied to the formula $A \supset B$. Thus this attempt at proving the original sequent (3) must be abandoned. The student can do this simply by clicking some of the sequents inside the proof (4). For instance, clicking on the implication '$\supset$' on the right of the third line in proof (4) substitutes it with

$$\dfrac{\dfrac{A, (A \supset B)\&(B \supset C) \Rightarrow C}{(A \supset B)\&(B \supset C) \Rightarrow A \supset C}\, R\supset}{\Rightarrow ((A \supset B)\&(B \supset C)) \supset (A \supset C)}\, R\supset \tag{5}$$

and the student is back on his way towards a completed proof.

In their study, Aitken et al. (1998, Sections 5.1.4 and 5.4) did not find significant benefits in this kind of user interaction with a theorem-proving program. However, their audience were mostly seasoned users of the program, not students of proof theory in general. Hence the benefits they expected differ from ours.

3.1 Quantification

Adding the universal and existential quantifiers '$\forall$' and '$\exists$' to the proof theory in Table 1 is straightforward in theory, and is given in Table 2. However, we must also extend our "proof by pointing" principles to these new quantifier rules.

Table 2: The additional inference rules for the sequent system **G3i**.

We assume familiarity with the standard definition for the syntax of first-order predicate logic: variable symbols $x, y, z, \ldots$, terms $t, u, v, \ldots$ built out of these variable symbols and function symbols, and atomic formulæ built out of these terms and predicate symbols. In addition, $A[t/x]$ denotes the formula obtained by substituting the term t for every free occurrence of the variable symbol x in the original formula A. (Negri and von Plato, 2001, Chapter 4.1)

Then we add to the connectives and rules of Table 1 the following quantifiers and rules:

	left rule	right rule
$\forall$	$\dfrac{A[t/x], \forall x.A, \Gamma \Rightarrow C}{\forall x.A, \Gamma \Rightarrow C}\, L\forall$	$\dfrac{\Gamma \Rightarrow A[y/x]}{\Gamma \Rightarrow \forall x.A}\, R\forall$
$\exists$	$\dfrac{A[y/x], \Gamma \Rightarrow C}{\exists x.A, \Gamma \Rightarrow C}\, L\exists$	$\dfrac{\Gamma \Rightarrow A[t/x]}{\Gamma \Rightarrow \exists x.A}\, R\exists$

However, these new rules $R\forall$ and $L\exists$ have an additional *restriction* (also known as proviso or side condition) on their use: The variable symbol y substituted for x must not occur free in the conclusion below the line.

The first obstacle is the term t substituted for x in the rules $L\forall$ and $R\exists$. While the rules do permit the student to choose any term t he wants, he should choose such a term t which allows him to complete the proof later. This completion in turn requires suitable applications of axioms (2), where the form of the atomic formula P depends on the term t chosen now. Thus it would seem that choosing t needs more information than can be extracted from a

single mouse click, since t does not have to appear in the sequent below the line. Moreover, it seems that the student should know how to choose a good t even before he can see what kinds of axioms (2) he will later need to apply. Do we therefore suffer from the same drawback as rule (1) after all?

Fortunately we do not, because we can incorporate *metavariables* as Paulson (1996, Chapter 10.4) does in his sequent-based theorem prover with a command-line user interface. A metavariable is simply a kind of bookmark standing for some term whose actual form does not yet concern us and can be determined later. Then the student can still just click on a quantifier '$\forall$' on the left or on '$\exists$' on the right side of a sequent: the response is to create a previously unused metavariable for t when applying the corresponding rule of Table 2.

These created metavariables receive their eventual values when the student applies axioms (2). Now the sequent is of the form $P, \Gamma \Rightarrow P'$ where the atomic formulæ P and P' do not have to be exactly alike yet because they may contain metavariable occurrences. We can now reinterpret the axioms to require that P and P' must be made exactly alike by selecting suitable values for their metavariables. The student can then select the axiom he wants by *grabbing P by its propositional symbol with the mouse, dragging it over the '$\Rightarrow$', and dropping it onto P'* (or vice versa). Thus we refine our earlier user interface concept into drag-and-drop for axioms and point-and-click for other rules.

The standard way to make two expressions alike is *unifying* them (Paulson, 1996, Chapter 10.7) (Russell and Norvig, 2003, Chapter 9.2), and it suffices here as well. Once unification binds some metavariable χ into some value v, then v must be substituted for χ throughout the whole proof under construction. Hence quantification introduces global dependencies between different parts of the same sequent proof, whereas the effects of the propositional rules of Table 1 remained local to the sequents in each rule. On the other hand, assumption-handling already introduces global dependencies into the propositional rules of Natural deduction. Conversely, the binding of v into χ must be undone if the student later abandons this axiom application in favour of some other strategy.

Let us then turn to the remaining quantifier rules $L\exists$ and $R\forall$. At first sight their restriction seems innocuous: Just pick some previously unused variable symbol as the new y. But suppose that the sequent below the line contains the metavariables $\chi_1, \chi_2, \chi_3, \ldots, \chi_k$. (More precisely, they are the metavariables χ_i created in this branch of the proof.) Then the restriction actually demands that y must not occur in the values $v_1, v_2, v_3, \ldots, v_k$ that these metavariables will eventually have. The most common suggestion to ensure this seems to be picking a previously unused *Skolem* function symbol f_y instead, and using the term $u = f_y(\chi_1, \chi_2, \chi_3, \ldots, \chi_k)$ instead of y (Paulson, 1996, Chapter 10.7). Then any unification which tries to include u in a value v_i fails, because it would make v_i an infinitely long term, and that is prohibited. However, Paulson (1996, Chapter 10.7) maintains that Skolem functions make the formulæ unreadable, and this should be avoided in an educational tool. Instead he suggests recording the induced requirements "y must not occur in χ_i" and heeding them during unification. The downside of his suggestion is that this collection of induced requirements is a new aspect in our user interaction concept, which up to this point has consisted only of the proof itself, augmented with metavariables.

Either suggestion points to a more fundamental pedagogical problem: the unification fails when the axiom requested by the student does not apply. But how should the student be told *why* it failed? If the tool silently ignores the student's request and refuses to modify the proof, then the student gets no feedback at all and might even think that the tool is broken. On the other hand, the unification algorithm proceeds differently than the student, and therefore the position at which the algorithm detects the impossibility of the request may be meaningless to the student. The problem of generating meaningful error messages when unification fails is also of independent interest, since unification is used in for example compilers of such high-level programming languages as Standard ML (Paulson, 1996). However, the criteria for meaningfulness may also differ in logic and programming.

Note finally that the quantifier treatment suggested here is compatible with the "peeling off the outermost box" computation suggested above: the student can click inside a quantified formula, and the quantifiers enclosing the clicked position can be peeled off automatically with these rules.

3.2 Equality

Using the equality $t = u$ of two terms t and u is so common in proofs that it warrants its own special treatment. One possibility is having special inference rules (Negri and von Plato, 2001, Chapter 6.5)

$$\frac{t = t, \Gamma \Rightarrow C}{\Gamma \Rightarrow C}\ \mathit{Ref} \qquad \frac{P[u/y], t = u, P[t/y], \Gamma \Rightarrow C}{t = u, P[t/y], \Gamma \Rightarrow C}\ \mathit{Repl}$$

for reasoning about equality. We extend our user interaction concept to these rules as follows.

The intuition of rule *Repl* is that the occurrence of term t at position y within the atomic formula P should be replaced with u. We can refine the drag-and-drop approach introduced in Section 3.1 by allowing the student to grab not only whole atomic formulæ but also their terms and subterms. Then the student can simply drag the subterm at position y onto the left-hand term t of the equality to use (or vice versa). Again, these terms can contain metavariables, and they are therefore unified first.

Rule *Ref* resembles the rules $L\forall$ and $R\exists$ in Table 2 in the sense that it too introduces some previously unknown term t above the line which has no counterpart below the line. The difference is that whereas rules $L\forall$ and $R\exists$ provided the quantifier for the student to click, rule *Ref* offers no such visual disambiguating element. However, we can introduce such an element, if we make a mild additional assumption on the way the student is likely to proceed: We expect that he will use the new equality $t = t$ in the next step. Then we can also expect that the term t appears already as a subterm below the line. Thus we can refine our point-and-click approach as follows: clicking on a subterm t means an application of rule *Ref*, whereas clicking on a predicate symbol or a connective refers to the rules in Table 1 as before.

Unfortunately these suggestions do not permit using these rules automatically, in contrast to the suggestions in Section 3.1.

4 Implementation

We have a prototype implementation of the proof-by-pointing approach outlined in Section 3. This prototype is called ProEd for PRoof EDitor. Although the presentation of Section 2 focused on **G3ip**, an intuitionistic logic, the implementation supports the related classical logic **G3cp** (Negri and von Plato, 2001, Chapter 3) as well.

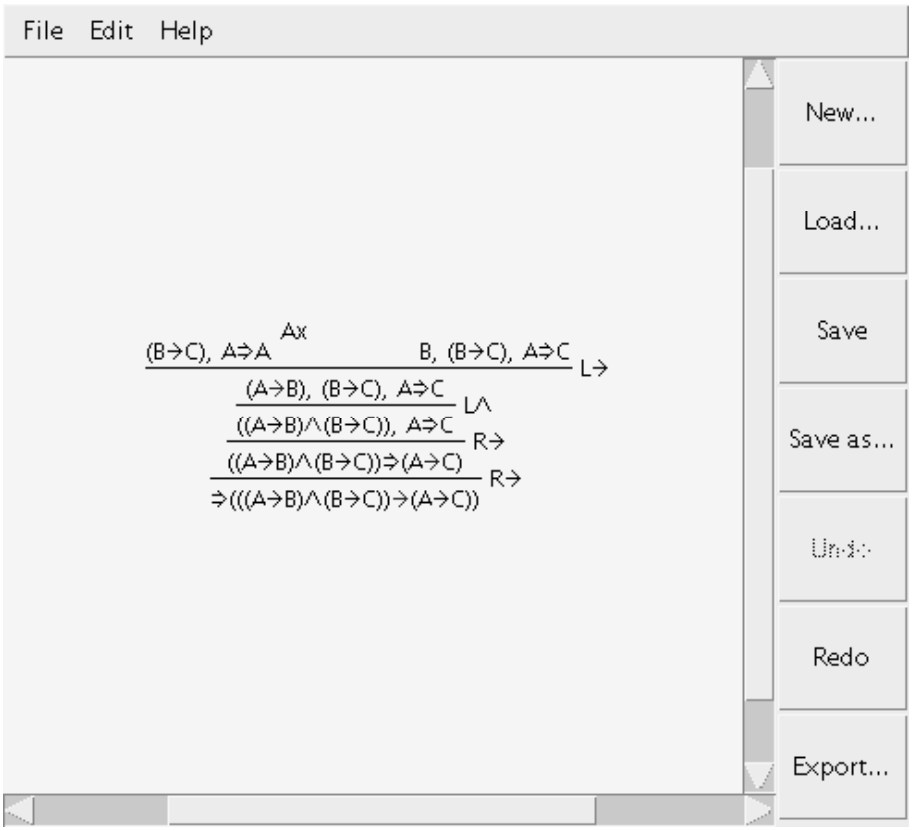
Figure 1 presents a screen shot of the tool. The sequent to prove is (3). The student needs only *two* mouse clicks to get this far: The first click on the implication '$\supset$' between A and C yields the incomplete proof (5), and the second click between A and B yields the situation as shown.

Our current implementation restricts itself to propositional logic. That is, it does not support the refinements suggested in Sections 3.1 and 3.2 yet. On the other hand, the restriction to propositional logic also permits some extra features such as automatic detection of axioms. In addition, our implementation provides facilities to save an incomplete proof and continue it later, or to print it or parts of it in LaTeX.

Adding the refinements suggested in Section 3.1 would be the next step in developing our implementation further, because then it would cover all the core material of the introductory proof theory course mentioned in Section 1. The further refinements in Section 3.2 would be straightforward after that. Systems like ETPS (Andrews et al., 2004) and STRATUM (Janhunen et al., 2004) also offer pedagogical tools for assessing the students' progress, which our ProEd currently lacks.

We have offered our implementation to the students of the course. Anecdotal evidence suggests that they preferred it to the PESCA tool also mentioned in Section 1, but no formal comparisons have been performed. This preference was to be expected anyway, since users tend to generally prefer graphical to textual user interfaces (at least until they become more proficient in the task at hand). The course is elective, which means that it may not be offered regularly, and that its attendance may be small. These aspects of the course make it harder to gather more substantial evidence.

The current implementation is wholly written in the Java programming language, and therefore runs on multiple platforms. Bornat and Sufrin (1997) have developed JAPE which both supports similar functionality and is customizable to different proof theories. Thus JAPE is an example of a meta-framework in the sense of Section 1, one with its emphasis on the user



The tool differs slightly from the proof system in Table 1 in the following three respects:

1. Axioms are explicitly marked with 'Ax' to provide an explicit visual clue that this branch has now been dealt with.
2. Conjunction is denoted with '$\wedge$', disjunction with '$\vee$' and implication with '$\rightarrow$'.

   This notation is perhaps more common in mathematics and computer science, while the former seems to be preferred by philosophers.
3. The left implication rule $L\rightarrow$ omits displaying the repeated formula $A \rightarrow B$ to save screen space. Invoking this rule again would merely lead into an infinite regress, as discussed in Section 3.

Figure 1: A screen shot of the PROED tool.

interface. Switching from our proprietary implementation into a JAPE-based one would entail encoding the rules in Table 1 in its logic encoding language, together with enough guidance information for the rule selection machinery of Section 3 to work. Future versions of our tool may well be JAPE-based, if it turns out to suit our needs. The support for quantifier handling in JAPE would be especially valuable in implementing the refinements suggested in Sections 3.1 and 3.2. On the other hand, JAPE seems to offer no pedagogical tool support, so choosing the next implementation platform involves a tradeoff.

The implementation described here is available at `http://www.cs.helsinki.fi/group/toto/`.

5 Conclusion

We have explained the principles behind PROED. This computerized tool is intended for novice students of propositional sequent systems of logic. The tool frees the student from tedious but necessary bookkeeping aspects, and lets him concentrate on exploring the different choices available for proof construction. Such exploration is made especially easy by providing a graphical user interface where single mouse clicks suffice for disambiguating between these choices.

Acknowledgements: The author wishes to thank Raul Hakli for fruitful discussions; he has been both the course assistant and a draft version reviewer of the textbook (Negri and von Plato, 2001). Thanks are also due to the implementors of the prototype in Section 4: team leader Marja Huovinen, and participating students Teppo Kankaanpää, Jaakko Nenonen, Aki Nyrhinen, Visa Röyskö and Juha-Matti Tapio.

References

Aitken, J., Gray, P., Melham, T., Thomas, T., 1998. Interactive theorem proving: An empirical study of user activity. Journal of Symbolic Computation 25, 263–284.

Andrews, P., Brown, C., Pfenning, F., Bishop, M., Issar, S., Xi, H., 2004. ETPS: A system to help students write formal proofs. Journal of Automated Reasoning 32, 75–92.

Bertot, Y., Théry, L., 1998. A generic approach to building user interfaces to theorem provers. Journal of Symbolic Computation 25, 161–194.

Bornat, R., Sufrin, B., 1997. Jape: A calculator for animating proof-on-paper. In: 14th International Conference of Automated Deduction (CADE 14). Vol. 1249 of Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 412–415.

Carpenter, B., 1997. Type-Logical Semantics. The MIT Press.

Janhunen, T., Jussila, T., Järvisalo, M., Oikarinen, E., 2004. Teaching Smullyan's Analytic Tableaux in a Scalable Learning Environment. These proceedings.

Kapur, D., 2004. Preface to a special issue on "automated reasoning and theorem proving in education" (arte). Journal of Automated Reasoning 32, 1–2.

Negri, S., von Plato, J., 2001. Structural Proof Theory. Cambridge University Press.

Paulson, L., 1996. ML for the Working Programmer, 2nd Edition. Cambridge University Press.

Restall, G., 2000. An Introduction to Substructural Logics. Routledge.

Russell, S., Norvig, P., 2003. Artificial Intelligence: a Modern Approach, 2nd Edition. Prentice-Hall.

Teaching Smullyan's Analytic Tableaux in a Scalable Learning Environment

Tomi Janhunen, Toni Jussila, Matti Järvisalo, and Emilia Oikarinen
Helsinki University of Technology
Tomi.Janhunen@tkk.fi

Abstract

This paper describes an automated learning environment in which various kinds of formal systems such as logic, automata and grammars can be taught on basic courses in (theoretical) computer science. A central design criteria of the system is scalability as hundreds of students are expected to participate in courses which are using the system to automate their home assignment processes. High numbers of students imply the need for automated, partially randomized assignment generators in order to create challenging problem instances for students as well as to minimize plagiarism. In this paper, we consider the case of Smullyan's tableau method as a concrete example and discuss in further detail the design of a generator for logical problems which are to be solved by the student. The non-deterministic nature of tableau proofs is identified as a factor that makes the automated generation of assignments much harder than generating assignments which are solvable just by following some deterministic algorithm. Moreover, the minimum number of branching nodes in the respective tableau proof is proposed as a criterion for measuring the quality of a problem instance. The same criterion is then used to re-evaluate tableau proofs that were submitted by our students in years 2000–2004.

1 Introduction

It is challenging to teach computer science students theoretical issues in spite of the fact that the students have some background in mathematics and disciplines required in programming. The same can be stated about teaching them how to solve logical problems. In our case, this challenge is faced on a course "*Logic in Computer Science: Foundations*" (Janhunen, 1995–2004)[1] at Helsinki University of Technology (TKK). The course is mainly targeted at third year students in the degree programme of Computer Science and Engineering (CS&E) and it has appeared in the curriculum of TKK since the eighties. The course contents were very traditional at first but a notable shift took place when a book by Nerode and Shore (1993) was introduced as the course book. Rather than using classical proof systems, the book concentrates on using Smullyan's analytic tableaux (Smullyan, 1968) as the main proof method. A nice feature of the method is that a concrete counter-example can often be constructed in cases where proofs turn out to be nonexistent. Detecting as well as showing unprovability becomes much more difficult if a classical proof system is used.

In addition to explaining students general proof strategies, we find it very important that students get hands-on experience in writing tableau proofs. To ensure this, we decided to include a series of home assignments as a part of the course requirements. In the beginning, the assignments were distributed on sheets of paper to the students who then returned their answers for the lecturer's manual inspection. In the late nineties we realized the potential of the World Wide Web (WWW) as a distribution channel and started the automation of home assignments by putting them available for download on the WWW.

The number of students taking our logic course rose from roughly 100 to over 400 a year during the nineties because the annual intakes to degree programmes related to information technology and telecommunications (like ours) were substantially increased. To address the problems created by the flood of students, the first author conducted a student project in summer 2000. The goal of the project was to automate the home assignment process as far as possible. Retrospectively speaking, the project achieved its goal very well. The system that resulted is called STRATUM and it provides an automated learning environment where

[1] The authors of this paper form the team responsible for the course in the academic year 2003–2004.

— personal home assignments are automatically generated for students,

— home assignments are put available for download on the WWW,

— students are provided automated tools, such as graphical editors and theorem provers, for doing/solving their home assignments,

— the tools deliver the answers of students for approval using electronic mail, and

— the answers of the students are checked automatically a few times every day using assignment-specific verifiers such as proof checkers or theorem provers.

In this paper, we overview the current system and discuss its potential as an automated teaching tool as seen from both the teacher's and the students' points of view.

In particular, we are interested in the problem of generating *personal* home assignments for hundreds of students, which is often necessary to hinder students from copying answers from each other. This becomes a problem of some sort if the number of different assignments is small compared to the number of students. To this end, we have designed a partially randomized scheme for creating challenging logical problems to be solved by students using Smullyan's tableau method. The *non-deterministic* nature of tableau proofs, i.e. one has to make real choices when constructing them, makes the generation of suitable problem instances much harder. In this paper, we want to distinguish such problems (or assignments) from *deterministic* ones which are basically solvable by following some deterministic algorithm. Determinism eases the automated generation of assignments because the existence of a solution is usually guaranteed and it is enough to produce a variety of inputs for students.

However, assignments that are produced in some sense randomly are not necessarily satisfactory. Such assignments may appear counter-intuitive at first sight as they typically lack a real-world counterpart. Perhaps more importantly, randomness makes assignments often all too easy from the problem solving perspective, which in turn may frustrate students. Thus, ensuring the quality of automatically generated assignments in this respect becomes a problem of its own. One way to approach this problem is to simulate the student's problem solving process and in this way to estimate the level of difficulty of assignments generated according to a particular scheme. It is clear that non-determinism makes the simulation part more difficult by bringing the nature of a search problem into it. In the case of Smullyan's analytic tableaux, our proposal is to count the minimum number of *branching nodes* when assessing the relative difficulty of tableau proofs. This is a computationally very expensive (exponential) task in the worst case but still realistic for tableau assignments of the sizes that we consider appropriate for our students.

The rest of this paper is organized as follows. Section 2 gives an overview of the automated home assignment system STRATUM (Aho et al., 2000–2004). The main components of the system are identified and discussed further in the case of Smullyan's analytic tableaux. In Section 3, we introduce the number of branching nodes as a criterion for assessing the quality of tableau proofs. Based on this, the respective (average) qualities of tableau proofs submitted by our students in years 2000–2004 are determined. An analysis on the strengths and weaknesses of the system follows in Section 4 together with an analysis of students' feedback. Some related systems are addressed in Section 5. We focus especially on the nature of assignments, i.e. whether they appear deterministic or non-deterministic from the students' point of view. Finally, Section 6 concludes this paper. At the moment, there are also other courses utilizing the STRATUM infrastructure — demonstrating its flexibility. The current variety of assignments based on our architecture is also very briefly summarized.

2 Overview of the System

The architecture of STRATUM is shown in Figure 1. The main components are (i) an assignment generator, (ii) a (graphical) editor for composing answers, (iii) a verifier for detecting

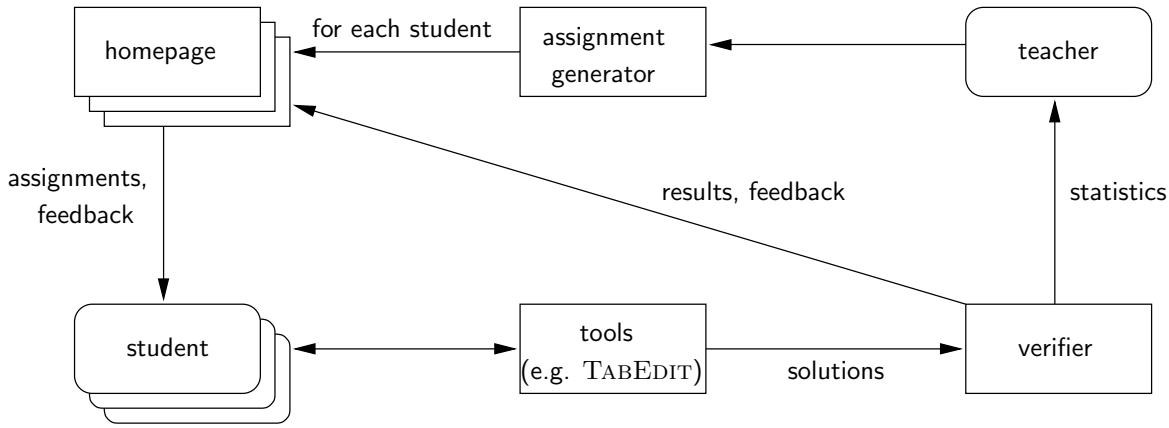


Figure 1: The overall architecture of the STRATUM system

correct answers, and (iv) a feedback channel for reporting any findings to the student.

The distribution of the home assignments to students is carried out through a personal WWW page and a home directory which are created automatically for each student in the beginning of the course. The assignment generator is responsible for creating a set of personal home assignments for students and linking them to their respective WWW pages. In certain cases, the automated generation of assignments is time-consuming for technical reasons. This implies that assignments have to be created off-line and a random choice among existing assignments is performed whenever a new set of assignments is needed. The student's interface to the system consists of the student's individual WWW page and the tools provided for solving the assignments. Depending on the assignment in question different tools can be used. The solutions are submitted for approval via electronic mail, a feature that is often integrated to the tools directly. Incoming submissions are transferred automatically to students' personal home directories and linked to their personal WWW pages.

A verifier is used to check if the answer/solution submitted by the student is correct. The verification of the submitted home assignments is done as a batch run a few times a day. The feedback from the verifier is linked to the student's WWW page. Thus, in addition to assignment distribution, the WWW page is also used as a feedback channel. The complexity of verifying the correctness of the solutions depends heavily on the problem type. In some cases the verdict can be definitive but there are assignment types for which only *partial correctness* can be established. Imagine, for instance, that the answer submitted by the student is an algorithm solving a specific problem. Probably the best we can do is to verify the correct output of the algorithm given certain problem instances as input. This remains very far from establishing the total correctness of the algorithm, i.e. correct outputs for all possible inputs.

In the following we discuss in more detail the components of the system in the case of teaching students to write proofs using Smullyan's analytic tableaux.

2.1 Creating Challenging Logical Problems

We generate valid propositional sentences in three phases, each of which is implemented as an *answer set program*; see (Niemelä, 1999) for an overview of answer set programming and (Simons et al., 2002) for a technical account of a state-of-the-art implementation SMODELS.

In the first phase, we form propositional sentences consisting of at most n propositional atoms and m connectives, such as $A \rightarrow (A \vee \neg B)$ when $n = 2$ and $m = 3$. These sentences can be viewed as candidates whose quality is checked in the latter two phases of assignment generation. The candidates passing the test implemented in the second phase qualify as *valid* propositional sentences. Such sentences are already guaranteed to have a tableau proof but the complexities of proofs can vary substantially. For instance, sentences of the form $\psi \vee \neg\psi$ where

ψ is any propositional sentence have a very simple tableau proof consisting of four nodes. In order to exclude sentences that have very short tableau proofs, we need an additional phase in the problem generator. In the third phase, it is examined whether the sentence ϕ given as input has a tableau proof of *branching depth* d which is defined as the maximum number of branching nodes on paths leading from the root node of a tableau to the leaf nodes of it. By increasing d until the existence of a tableau proof can be established, we get an estimate how complex the simplest proofs for ϕ are. This provides us the basis for selecting the most demanding valid propositional sentences. For instance, the tableau proofs for the validity of $A \leftrightarrow A \wedge (C \leftrightarrow (B \leftrightarrow (B \leftrightarrow C)))$ are of branching depth 5. This is mainly due to the high number of logical equivalences involved.[2] By excluding the occurrences of logical equivalence altogether, we obtain a class of assignments which is perhaps more interesting in the logical sense. Let us just mention $A \rightarrow ((A \rightarrow B \vee (A \rightarrow B)) \rightarrow B)$ as an example.

Sentences generated as described above are ready to be distributed to students as home assignments of propositional logic. Further ideas are needed in the case of predicate logic. However, we can utilize a propositionally valid sentence ϕ in this case, too. By replacing the atomic propositions of ϕ with atomic formulas and by adding a suitable quantifier prefix, we obtain a first-order sentence $Q_1x_1 \ldots Q_nx_n\phi$ in *prenex normal form* (Nerode and Shore, 1993). By applying quantifier rules backwards, we can push the quantifiers in $Q_1x_1 \ldots Q_nx_n$ into the structure of ϕ. As a result, we obtain valid sentences of predicate logic, like (1), which are directly usable as home assignments for the students. The multitude of quantifiers enforces a well-thought-out use of the most demanding tableau rules dealing with quantifiers.

$$\forall v \forall z \exists s \exists w((\exists r \exists t\, P(r,s,t) \rightarrow \\ ((\exists u\, Q(u,v) \rightarrow P(v,s,w)) \rightarrow \forall x\, R(x,w))) \rightarrow (P(s,s,w) \rightarrow \forall y\, R(y,z))) \quad (1)$$

2.2 Tableau Editor TabEdit

Our students are provided a graphical editor for creating analytic tableaux as answers to their home assignments. The TabEdit tool has been implemented by Marco Gaus and it supports both propositional and first-order sentences.

A sample assignment solved by a student using TabEdit is shown in Figure 2. The root node(s) of the tableau can be either imported from a text file in the internal format of TabEdit or written down directly using the program. Once the root nodes have been loaded, the student can extend the tableau simply by selecting nodes and applying appropriate tableau rules. If a rule is applicable to the node chosen by the student, new elements are generated and inserted into the tableau. Otherwise, an error message is displayed for the student and no further processing can be done for a specific amount of time. To make the trial-and-error approach infeasible, the time penalty increases exponentially with each wrong choice. In the end, the student marks each branch of the tableau either contradictory or non-contradictory.

At any point the formed tableau can be exported either in the internal textual format or in LaTeX format. An exported tableau can be imported back to the tool and thus it is possible to finish the tableau later on if desired. The submission procedure has been directly integrated to TabEdit so that finished tableaux can be easily sent for inspection.

2.3 Using a Proof Checker as a Verifier

Determining the validity of a given sentence can be a very challenging problem.[3] However, once a proof has been found, it is relatively easy to verify. A nice feature of a tableau proof is that it gives an affirmative answer to the validity problem being solved by the studen, so there is no need to further test/analyze the student's answer by any means.

[2] The tableau rules associated with logical equivalence (i.e. $\leftrightarrow$) branch a tableau proof unconditionally.

[3] The problem is **coNP**-complete for propositional sentences and semi-decidable for first-order sentences.

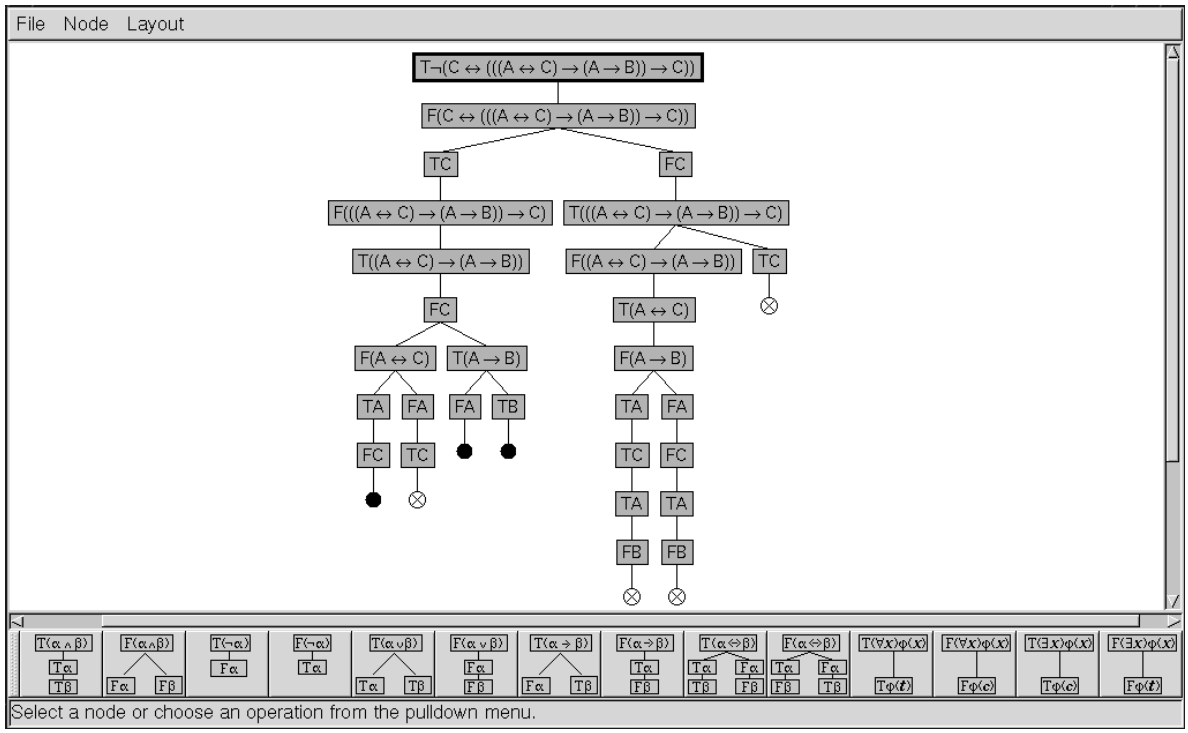


Figure 2: An assignment that has incorrectly been solved by the student

We use a proof checker called CHECKKER implemented by Vesa Norrman in Standard ML (short for *Meta Language*). Basically, a tableau proof like the one given in Figure 2 can be verified by passing through its nodes in order to check whether the conditions associated with tableau rules are met. A further task is to see that all paths of the tableau have been correctly marked as contradictory or non-contradictory ones as the detection of contradictions remains as the student's duty in addition to applying the tableau rules depicted at the bottom of the user interface in Figure 2. If a submitted proof does not qualify as a tableau proof for one reason or another, an error message such as the one given in Figure 3 is delivered to the student. In this case, the student's next task is to revise and resubmit the proof.

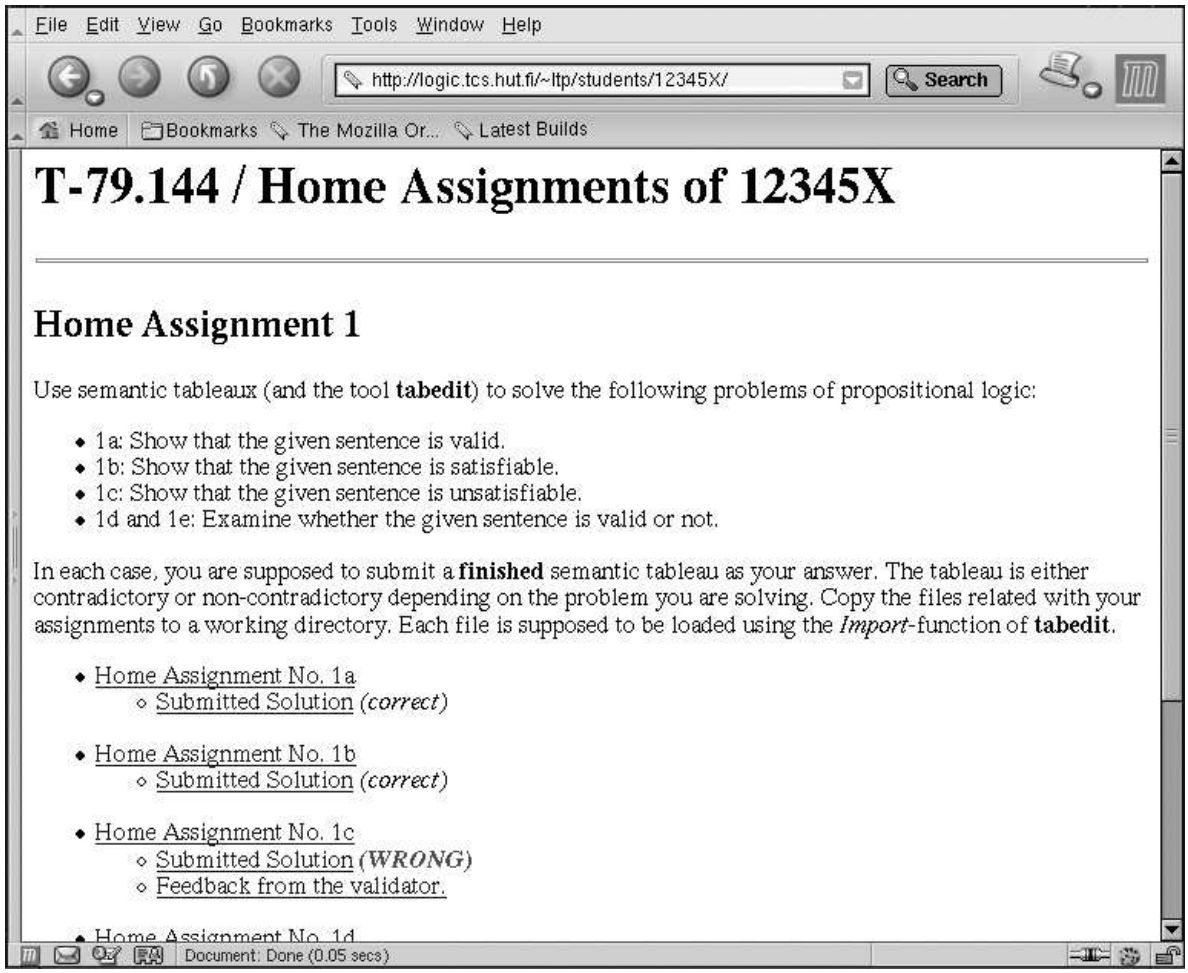
2.4 Assignment Distribution and Feedback

Each student has a personal password-protected WWW page under STRATUM. The page is automatically created for each enrolled student when the course begins. An example of a student's WWW page is shown in the upper part of Figure 3. The page is used to distribute the home assignments to the student during the course. When the student has submitted his/her solution to the verifier, the WWW page is updated to contain information about the submission. Thus the student knows right away whether the submission procedure succeeded. After the verifier has checked the submission, the resulting feedback for the student is linked to the student's WWW page. An example of feedback is shown in the lower part of Figure 3.

3 Measuring the Quality of Tableau Assignments

As discussed in the introduction, ensuring the quality of automatically generated assignments becomes easily a problem of its own. It is studied in this section whether the *number of branching nodes* in tableau proofs provides a useful measure in this respect. This is a slightly different measure compared to the one that was used when our tableau assignments were originally generated in 2000 (recall the notion of *branching depth* from Section 2.1).

More formally, we write $\mathrm{BN}(\tau)$ for the number of branching nodes in a tableau τ. For




```
Path 1 (from left) is contradictory however it is not marked contradictory
Path 3 (from left) is contradictory however it is not marked contradictory
Path 4 (from left) is contradictory however it is not marked contradictory
The tableau is contradictory
```

Figure 3: The student's WWW page which includes a hyperlink to an error message generated by a verifier for the assignment/solution shown in Figure 2.

instance, this number is 6 for the tableau shown in Figure 2. It is also possible to count the *minimum* number of branching nodes $\mathrm{MinBN}(\tau_0)$ where τ_0 refers to the root node of τ.[4] Note that $0 \leq \mathrm{MinBN}(\tau_0) \leq \mathrm{BN}(\tau)$ holds for all tableaux τ. It is computationally very demanding to count $\mathrm{MinBN}(\tau_0)$ as determining this number requires us to construct tableaux having an increasing number of branching nodes until a finished tableau starting from the root node τ_0 is found. Nevertheless, this process is guaranteed to terminate in the case of propositional logic. We have implemented this procedure using the SMODELS system mentioned in Section 2. The performance of the implementation is sufficient to cover the types of assignments we have been using so far.

To assess the measure sketched above, we re-evaluated all correct tableau proofs that were submitted by our students in 2000–2003. For each tableau τ, we calculated $\mathrm{MinBN}(\tau_0)$ and $\mathrm{BN}(\tau)$ in order to check whether τ has the minimal number of branching nodes. The number $2 \times \mathrm{BN}(\tau) + 1$ gives us the number of equivalence classes of non-branching nodes in τ and

[4]Here τ_0 can be viewed as the assignment supplied to the student while τ might be his/hers answer.

Table 1: Ranking accepted submissions received in 2000–2004: the percentages of non-minimal (NM) tableau proofs and the average relative biases (RB) w.r.t. the minimum.

Assignment		1a		1b		1c	
Year	n	NM (%)	RB (%)	NM (%)	RB (%)	NM (%)	RB (%)
2000	400	11	12	49	20	48	45
2001	266	15	12	56	26	55	46
2002	248	12	9	54	23	59	54
2003	167	13	17	54	23	52	44
2004	135	15	10	39	16	36	30

hence a rough estimate of the tableau size. Consequently, the ratio

$$\frac{2 \times \mathrm{BN}(\tau) + 1}{2 \times \mathrm{MinBN}(\tau_0) + 1} = \frac{\mathrm{BN}(\tau) + 0.5}{\mathrm{MinBN}(\tau_0) + 0.5}$$

describes the relative bias in the size of τ compared to the minimum — even when $\mathrm{MinBN}(\tau_0)$ equals to 0. The results are summarized on the four lines of Table 1 corresponding to years 2000–2003. As it is clear by the second column, the number of students is already dropping, although the number of passed students (i.e. n) for 2004 will still rise until the end of the term (and the final deadline for home assignments) is reached. The percentage of non-minimal tableau proofs is low for simple hand-made assignments (1a) but roughly 50% for automatically generated ones (1b and 1c), which consequently appear to be more demanding.

Inspired by the relatively high number of non-minimal proofs, we decided to carry out an experiment with the students that enrolled our course in autumn 2004. We pointed out the minimality aspect in the instructions given for the students and specifically asked them to write down as small tableau proofs as possible. Students were also shown a pair of proofs, one of which was minimal while the other had unnecessary branching nodes. The effect of the refined instructions can be seen from the last line of Table 1: the numbers of branching nodes became lower (except for 1a which is relatively easy due to hand-made assignments) and the deviations from the minimum were smaller on the average.

According to data summarized in Table 1, the correlation coefficient for $\mathrm{MinBN}(\tau_0)$ and $\mathrm{BN}(\tau)$ is as high as $\rho = 0.8219$. This suggests that the minimum number of branching nodes provides a reasonable way to measure how demanding tableau assignments are. We checked one of the examinations arranged in 2003 to see if there were any correlation between $\mathrm{BN}(\tau)$ and the number of points granted for τ but the result turned out to be negative. We believe that this is due to the fact that students are not asked to pursue minimal tableau proofs in examinations and the evaluation principles neglect minimality altogether. This is mainly because we fear that assignments would become too difficult for the students if we insisted on minimality. On the other hand, by taking the minimality aspect into account as far as possible, students can learn a good heuristics for writing tableau proofs.

4 SWOT Analysis and Student Feedback

We proceed by performing a brief SWOT analysis of the learning environment both from the students' as well as the teacher's perspective, and discuss some feedback given by students during 2000–2003.

Strengths. Providing a graphical interface for solving home assignments and visualizing abstract concepts, such as analytic tableaux and automata, helps especially visual learners, i.e. students who learn best by visual stimulation. We find this especially true in the task of

teaching theoretical concepts to more practically oriented students. As submitted solutions are logged by our system, additional guidance can easily be given. Furthermore, the students can choose when and where to solve the assignments. It should also be noted that STRATUM relieves the teaching staff from the laborious but still relatively easy-to-automate task of checking submitted answers.

Weaknesses. While plagiarism is made impossible by providing individual home assignment problems, little can be done to ensure that a particular student has solved the problems him-/herself. The system must be heavily tested in order to minimize the number of bugs in the system as any malfunctions of the systems or incorrect feedback to students can easily discourage and irritate them.

Opportunities. The flexibility and extensibility of the STRATUM infrastructure allows for easily integrating new types of assignments in the future. Furthermore, applying STRATUM on different courses (homework topics) is also possible.

Threats. Failures in system infrastructure lead to unavoidable deadline extensions. The system must also have a proper backup mechanism to avoid data loss in case of e.g. disk failures. Changes in the curriculum or the course contents may turn certain assignments outdated or even useless — implying further revisions to them, or eventual loss of resources spent on implementing them initially.

We also provided the students with a possibility of giving feedback using an HTML form. The form consists of a fixed questionnaire and some extra space reserved for the student's personal opinions. In addition, students can send electronic mail to course personnel and discuss their problems in the respective news group. When asked to grade the electronic learning environment using a scale from 1 to 5, the average was 3.09. Many students considered the graphical tool as a good way to learn about Smullyan's tableau method. In the students' personal feedback, the following opinions occurred most frequently.

— Some students claim that solving the exercises with TABEDIT is too easy in the case of propositional logic.

— In the case of predicate logic, students tend to send in complaints with an incorrect analytic tableaux (e.g. an unfinished one) attached. Then they claim unjustifiably that the tableau must be correct and that something is wrong with the verifier.

— Students are sent passwords in order to grant access to their personal WWW pages. Unfortunately, they tend to forget or delete their passwords accidentally and passwords have to be resent manually in the present version of the system.

Considering the first item, propositional logic exercises can be solved using a rather simple algorithm (although in that case the size of a proof may greatly increase), which is figured out by many students. This is, however, not the case for predicate logic due to the instantiation of variables. Also, requiring minimal proofs would circumvent this problem. As for the second item, the negative feedback is typically due to lack of understanding the tableau proof construction. Thus, exercises in predicate logic fulfill their purpose as it is typically not possible to solve them without referring to the course material. Regarding the last item, an automated mechanism could easily be created for password inquiry.

5 Related Work

Considering using TABEDIT as a tool for teaching tableau proofs, the most related system that we are aware of is WINKE (Endriss, 1999). Similarly to TABEDIT, WINKE provides a graphical editor for constructing proof trees. However, apparently WINKE does not provide the infrastructure for managing the submission and evaluation of assignments in a large scale. The WINKE system is a self-standing software tool, i.e. it is installed completely under Windows on the student's own computer. Rather than Smullyan's analytic tableaux, WINKE

is based on the KE calculus (D'Agostino and Mondadori, 1994) which combines features form Gentzen's natural deduction (notably an analytic cut rule) with Smullyan's tableaux. A further difference is that additionally WinKE has an interactive proof assistant that suggests which rules to apply next. This feature can be disabled in educational situations.

Next we would like to point out (dis)similarities between STRATUM and another framework called TRAKLA2 (Korhonen et al., 2003), which is an automatic learning environment for data structures and algorithms. The TRAKLA2 system allows students to do exercises related with data structures by manipulating them through a graphical interface. Additionally, TRAKLA2 includes a visualization component which animates a model solution when requested. Perhaps the most notable similarities between STRATUM and TRAKLA2 are that (i) both systems have been developed for courses with hundreds of enrolled students, (ii) graphical user interfaces are used for visualization, and (iii) solutions are submitted to a centralized server which then evaluates the submitted answers and gives feedback to the students. On the other hand, TRAKLA2 contains some functionalities that are not present in STRATUM. For example, in addition to saving the solution submitted by a student, TRAKLA2 contains the ability to log highly precise *interaction* data about what the student does (i.e. pointer clicks etc.) when solving a given problem. This gives means to analyze students' learning processes in detail. Yet another ascpect of TRAKLA2 is that the student may view a model solution to the problem at hand when he/she wishes to do so. As a consequence, the student's assignment is replaced with a new one. Such functionalities could be integrated to the STRATUM system, too.

There is, however, a notable difference between STRATUM and TRAKLA2 in the *exercise generation process*. As noted in the introduction, the *non-deterministic* nature of the tableau method considered here makes the generation of suitable assignments a nontrivial task. Input data for assignments involving e.g. basic data structures can essentially consist of any set of randomly generated values, possibly combined with rather simple checks on the values such as removing duplicates. Moreover, given specific input data, the process of solving the associated problem is deterministic as the student has to only follow a specific algorithm. In contrast, in the case of tableaux the input data is a logical sentence. The difficulty of proving whether a sentence is valid does not depend on its length as such. Therefore, the hardness of constructing a proof for a random sentence must be checked. This process is much more involved. Moreover, a unique tableau proof for a given sentence does not exist in general, and the student is forced to make real choices during the proof process. Depending on the ingenuity of the choices made, the resulting proofs can vary greatly in size.

6 Conclusions

In this paper, we describe a general infrastructure for implementing automated home assignments for students enrolling on courses in theoretical computer science and related fields. The STRATUM system has been operating since autumn 2000 and thousands of home assignments are submitted and checked each term. On the basis of our experiences, we conclude that the system scales well for courses taken by hundreds of students. The use of the system enables us to avoid a manual inspection task of 10 person weeks (assuming 400 enrolled students, 10 assignments, an average resubmission rate of 100%, and an average inspection time of 3 minutes) in the case of the tableau proof assignments used in our logic course. This rough calculation gives us an idea about the rate at which we earn back the resources spent on a particular home assignment which is expected to remain in effective use for years.

We have not specifically evaluated the effect of automated home assignments on learning results but our impression is that our students are performing at least as well as before. According to the feedback that we have obtained, our students are satisfied with the types of tableau assignments that we are offering them. Nevertheless, it remains as our future work to explore new schemes for creating home assignments. One idea is to utilize the measure established in Section 3. The design of new schemes often leads to challenging research

problems, which we intend to solve using modern AI techniques like answer set programming.

Currently, the STRATUM system is additionally used on another course arranged by our laboratory at TKK; in summer 2002, the system was extended with a new set of home assignments designed for a basic course in theoretical computer science[5] taken by roughly 800 students every year. These assignments deal with regular expressions and finite automata.

Automating these two courses has been very crucial to us and our laboratory due to the high number of students and a shortage in financial resources. In the future we intend to improve the system and to extend it with new kinds of home assignments. Such an initiative was already taken in summer 2004 when we started to develop new home assignments for both courses; this time themes being the evaluation of first-order formulas in classical structures and context-free grammars, respectively. The system has also received attention from other units of the CS&E department at TKK: a compiler course organized by the Laboratory of Information Processing Science has introduced the system this autumn.

Acknowledgements

We would like to thank the team of students that have been involved in the development of the system: Pauli Aho, Janne Auvinen, Marco Gaus (Technical University of Clausthal), Jan Kallenbach (Technical University of Ilmenau), Vesa Norrman, Antti Stén, and Tommi Tykkälä. We are also grateful to Prof. Pekka Orponen who has participated in designing the system as well as instructing the students involved in the project. The development of the system has been partially funded by the study committee of Helsinki University of Technology.

References

Aho, P., Auvinen, J., Gaus, M., Kallenbach, J., Janhunen, T., Norrman, V., Oikarinen, E., Orponen, P., Stén, A., Tykkälä, T., 2000–2004. STRATUM – an infrastructure for implementing automated home assignments. The current system can be visited at `http://logic.tcs.hut.fi/`.

D'Agostino, M., Mondadori, M., 1994. The taming of the cut: Classical refutations with analytic cut. Journal of Logic and Computation 4 (3), 285–319.

Endriss, U., June 1999. An interactive theorem proving assistant. In: N.V., M. (Ed.), Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '99. Springer-Verlag, Saratoga Springs, NY, USA, pp. 308–312, LNAI 1617.

Janhunen, T., 1995–2004. Logic in computer science: Foundations. Consult `http://www.tcs.hut.fi/Studies/T-79.144/` for the WWW page of the course.

Korhonen, A., Malmi, L., Silvasti, P., 2003. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In: Proceedings of the 3rd Finnish/Baltic Sea Conference on Computer Science Education. pp. 48–56.

Nerode, A., Shore, R. A., 1993. Logic for Applications (1st edition). Springer-Verlag, Berlin.

Niemelä, I., 1999. Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25 (3,4), 241–273.

Simons, P., Niemelä, I., Soininen, T., 2002. Extending and implementing the stable model semantics. Artificial Intelligence 138 (1–2), 181–234.

Smullyan, R. M., 1968. First-Order Logic. Springer-Verlag, Berlin.

[5] See `http://www.tcs.hut.fi/Studies/T-79.148/`.

PART 3
Discussion paper presentations

Analysing Discussion in Problem-Based Learning Group in Basic Programming Course

Päivi Kinnunen, Lauri Malmi
Helsinki University of Technology
Laboratory of Information Processing Science

pakinnun@cs.hut.fi, lma@cs.hut.fi

Abstract

Teacher needs to know how students learn to be able to plan a good course structure. It is important to recognize, which issues students consider hard or what kind of misconceptions they have. This is especially true when problem-based learning (PBL) method is used. To better understand what happens in a PBL group and which topics students talk about we need to analyze the content of conversation during group meetings. In this study, we are especially interested in factual aspects of the conversation leaving the social aspect aside. We are planning to classify factual addresses in the talk so that we would be able to tell, for example, how much there is discussion about syntactic, semantic or schematic issues. This information would be valuable for the teacher who wishes to know how students' learning is progressing. Since we are using different kinds of problems to trigger the learning process in a PBL group, it would be interesting to analyse whether different problems trigger different talk, as well. This way we might get some evidence what kind of learning each type of problem promotes.

This is a discussion paper where we will present different categories for the classification we are planning to use, and some examples of the analysis.

1 Introduction

Problem-based learning (PBL) is a method of group learning that uses problems or cases as a starting point to the learning process (Schmidt, 1983). In our previous research we have studied interaction in a PBL group on an introductory programming course (Kinnunen, 2004; Kinnunen and Malmi, 2004). We have been concentrating on the tutors' effect on groups' work, groups' efficiency and students' feedback concerning the PBL method (Kinnunen, 2004). Analyses were performed at the group level. Now we want to advance our research to study PBL groups' factual speech in more detail and on a more individual level. With this approach we hope to gain more information about what students actually talk during group meetings, what are the subjects that students spend most time with, how much there are correct, vague or incorrect statements and how other students/tutor respond to those. This information would be useful when trying to gain better understanding about the learning process of computer programming. This would help teachers when planning courses and when educating new tutors to PBL groups.

2 Target of the study

Data is collected from students at Information networks curriculum at Helsinki University of Technology. In the first semester, students take an introductory programming course (in total 7 study weeks) that applies the PBL method. The course consists of PBL sessions, self-directed learning period, programming assignments, a personal programming project and an exam. In addition, students write essays and draw concept maps.

We are interested in a conversation during the PBL group meetings. Especially, the parts of conversation that concern facts or assumptions about programming are in the focus of our interest. Therefore we videotaped two groups' PBL sessions so that we got three opening and three closing sessions from each group.

3 Method

The conversation from the videotape is lettered into text. After lettering there are several possibilities to continue. We are planning to start our analysis by looking first at larger sections of conversation at the time and specifying the topics students are talking about. We assume that possible topics might be, for example, algorithms, programming concepts, application issues, programming process and problem solving. However, we stress that those previously mentioned topics are hypothetical ones and that in our research we look at what kind of topics emerges from the data.

After the first phase of the analysis we have a general picture of what students talk about during the group meeting. In the second phase we define our focus and analyze conversation on address level. Addresses are divided into different categories, which are composed of the topics collected in the first phase of the analysis. These categories can then be further divided into subcategories such as syntactic, semantic, schematic and conceptual issues. These subcategories would help us to pass on to more theoretical level in our analysis.

In the third phase of our analysis we look again at larger section of conversation and analyze how factual, facile or profound this section of conversation is. The fourth phase of the analysis is to investigate tutor's role in a group and students' strategies to deal with, e.g., facile or incorrect addresses. We want to know in what kind of situations tutors are needed. What are the topics that trigger the kind of conversation where tutor's knowledge is needed? For example, what are the issues that students have misapprehensions with in a way that tutor's intervention is needed. Our goal is therefore to describe different conversation sections (e.g. How incorrect address is received? How and by whom it is corrected?) This can be studied by using modified Bales Interaction Process Analysis (Bales, 1950/1951) and Flanders Interaction Analysis System (FIAS) (Flanders, 1965) We have used this method earlier to do research on the social aspects of group work (Kinnunen, 2004). With further modifications we can use the same method to meet our present goals.

The basic idea of this method is that the text is coded into numbered categories so that each address of the conversations is analyzed separately. This way we get a long sequence of numbers which is then transformed into a matrix. Transformation into a matrix is done so that the first number in a pair indicates the row and the second number indicates the column in the matrix. In this place we put a marker. The second pair is created so that the first number is the last number of the former pair and the second number is the third number in the sequence of numbers, and so forth. When the matrix is ready, we can easily see, which addresses follow each others, thus getting kind of interaction paths.

3.1 Categories

In our previous study we used the following categories (Kinnunen and Malmi, 2004):

1. Talks about something else than topic of the meeting
2. Releases tension
3. Encourages or agrees
4. Chairman's address
5. Gives his/hers own explanation about content
6. Student lectures to others
7. Asks a question
8. Responds to a question

9. Strongly disagrees, expresses negative feeling
10. Getting organised
11. Change
12. Silence
13. Confusion
14. Tutor lectures
15. Tutor responds to a question
16. Tutor comments generally

Now we modify the categories 5 - 8 so that we create subcategories to them. For example, category 5 (gives his/hers own explanation about content) has subcategories algorithms, programming concepts, applications, programming process. These subcategories are created from the topics we get from the first phase of analysis and therefore above-mentioned categories are hypothetical ones. An additional mark can be added if a statement is incorrect or vague. The same subcategories are created to categories 6, 7 and 8, too.

3.2 What kind of information it is possible to get from this data?

From this kind of data it is possible to get a lot of information. However, we concentrate on looking at the content (topics) and the level of the factual conversation. In addition, we look more closely at conversation/interaction paths, which reveals something about the learning process and tutor's role in a group. To be more precise, from text data, which is divided into topics, we can calculate the frequency of each topic (an address that relates to a certain topic) either at the group level or the individual level and thus see, which group members actually are active when they are talking about programming.

Finally, there are many situations where we assume that the conversation will be different. For example, since PBL groups get 9 – 10 different problems/cases during the course, we assume that different kinds of problems promote different kind of speech. Conversation is also probably different according to whether it was taped at the beginning or at the end of the course, as well. In addition, an interesting research question is whether the usage of a mind map during a PBL session will have any affect on factual speech. And if there is a difference what kind it will be?

4 Conclusion

Our goal is to gather information about students' learning by detailed analysis of student interaction in PBL groups. We are expecting to get information about what are the issues that arouse a lot of discussion, what components of programming skills are the ones that students have difficulties to understand and what kinds of misunderstandings they have during the learning process. Another aspect is that with this study we can detect essential parts of conversation where "learning takes place" or where we can see that students' way of perceiving things change. This kind of information would give some insight into the learning process. For the teacher it is essential to be aware of this process in order to be able to plan teaching so that it supports best students' learning.

References

Bales, R. F., 1950/1951. Interaction Process Analysis. A Method for the Study of Small Groups. Cambridge, MA: Addison–Wesley.

Flanders, N. A., 1965. Teacher Influence, Pupil Attitudes, and Achievement. Cooperative Research Monograph No. 12. U.S Department of Health, Education, and Welfare. Office of Education. Washington: U.S Government printing office.

Kinnunen, P., 2004. Interaction and Experiences in the University Student's PBL group. In Finnish. Vuorovaikutus ja kokemukset korkeakouluopiskelijoiden PBL-ryhmässä. Licenciate's thesis. University of Helsinki.

Kinnunen, P and Malmi, L., 2004. Some Methodological Viewpoints how to Evaluate Efficiency of Learning in a Small Group— a Case Study of Learning Programming. In Laine, A., Lavonen, J., and Meisalo, V. (Eds.) Current research on mathematics and science education. Proceedings of the XXI annual symposium of The Finnish Association of Mathematics and Science Education Research. University of Helsinki, Department of Applied Sciences of Education. Research Report 253.

Schmidt, H. G., 1983. Problem-based learning: rationale and description. Medical Education 17 (1), 11–16.

Statistical analysis of problem-based learning in theory of computation

Wilhelmiina Hämäläinen
Department of Computer Science, University of Joensuu, Finland

whamalai@cs.joensuu.fi

Abstract

Problem-based learning method has been applied to studying theory of computability in University of Joensuu during the last two years. In this paper, we analyze the data collected from two successive classes in 2003 and 2004 by common statistical measures. We compare the overall performance and satisfaction of problem-based and traditional learners, and analyze the influence of students' mathematical knowledge and gender in both methods.

1 Introduction

The main idea in *problem-based learning* (*PBL*) is to use problems, questions or puzzles as a starting point for learning. The problem is posed so that the students discover that they need to learn some new knowledge before they can solve the problem. PBL is thus a student-centered method which matches well with the *constructivist* view of learning (see e.g. (Ben-Ari, 2001)): it requires active processing of information, activates learner's prior knowledge, stimulates elaboration and organization of knowledge, and offers a meaningful context for learning. Several benefits have been observed (Lambrix and Kamkar, 1998; Kirsch, 1996; Greening et al., 1997), although negative results have also been reported (McCracken and Waters, 1999).

In computer science, the problem-based method has been typically applied only to programming and software courses (e.g. (Greening et al., 1997)), but in University of Joensuu, we have developed a problem-based method for a purely theoretical course, "Theoretical Foundations of Computer Science", which covers introduction to theory of computation.

According to our experiences, the results have been really successful: the students have committed themselves well and the dropout rate has decreased dramatically; the students have achieved very deep understanding of the subject measured by grades and quality of learning diaries; the experience has been enjoyable for both the students and the teachers; and finally, the method seems to support different learners very well. However, there are always students who react very critically against new experiments. They may have problems to adapt to new methods or the problem-based learning just does not suit for them. To help both students and teachers to make decisions on which approach to pursue, we have collected and analyzed data about our experiments. Especially, we aim to test the following hypotheses:

1. Mathematical background has a strong influence on student's success in theory of computation.
2. Problem-based learning attracts more female students. (Lambrix and Kamkar, 1998)
3. Poor students manage better in problem-based learning.
4. Problem-based learning prevents dropping out.

In the following, we will first briefly describe our course arrangements in Section 2. In Section 3, we will analyze the course performance: general grade distribution, comparison of two methods, and influence of mathematics and gender. In Section 4, we will draw the final conclusions.

2 Course description

During the last two years, we have tested the problem-based method for teaching theory of computability in the Department of Computer Science, University of Joensuu. A detailed description of our experiment can be found in (Hämäläinen, 2003).

In Joensuu, there is only one course on this subject, "Theoretical Foundations of Computer Science" (*TFCS*), which covers the theory of computability from finite automata and regular expressions to context-free grammars, pushdown automata, Turing machines and solvability issues. Both classes (TFCS-03 and TFCS-04) have been organized the same lecturer, but in TFCS-04, the foreign students were offered English lectures by an assistant teacher.

In our experiment we have adopted the seven step model, which is quite often used in the PBL education (David et al., 1999; Hakkarainen et al., 1999). The seven steps of each PBL cycle are represented in Table 6 (Appedix).

In both years, the course was given during 10 weeks, with four hours lectures and two hours exercise sessions per week. Half of the lecture time was used in opening and processing the problems in small groups according to the seven step model, and the other half was reserved for lecturing. A couple of times this normal program was replaced by playing problem solving games. During the group works, the students were tutored by the lecturer and volunteer assistants.

In the first half of the lecture the students first processed the last problem for about half an hour. Then they were given a new problem to be opened. They were also encouraged to meet the group members in the free time, and at least some groups had very active communication through chat. When the problem reports were returned we still had some general discussion with the whole group and the final conclusions were drawn.

In addition to the problem reports the students wrote *learning diary*, in which they were asked to process the learnt subjects, set questions, construct relations to previous knowledge, introduce their own applications, and especially reflect their own learning process. The learning diaries and the problem reports were evaluated, and the course grade composed of the points they have got in problems, learning diary and exercises together. The detailed evaulation criteria can be found on
`http://www.cs.joensuu.fi/pages/whamalai/tepe04/evaluation.htm`.

The students were given freedom to participate either problem-based learning or perform the course by traditional way with exercises and exams. However, the amount of lectures was only half of normal, and thus also the traditional way required some self-studying.

3 Course performance

3.1 Grade distribution

If the students chose PBL, the course performance consisted of problem points (50 %), exercise points (25 %) and learning diary points (25 %). In the traditional method, it consisted of exam points (75 %) and exercise points (25 %). The mean and standard deviation of total points are represented in Table 1a. We recognize that in PBL the total points are higher and deviation is smaller (i.e. majority has succeeded quite well), while in the traditional method the mean is lower but the deviation is larger (i.e. the results are more diverged). We assume that the talented students were more courageous to select the traditional way and managed excellently, but also the dropout rate was higher in the traditional way, as can be seen in Table 1b.

When we consider the overall grade distribution (Figures 1 and 2 in Appendix), we recognize that the problem-based learners' grades do not follow the normal distribution, but there are high peaks on the best grade. Obviously, problem-based learning encourages more students to deep learning. This was also recognizable during the course: many students in PBL were eager to study extra material and became real experts in the area.

Table 1: a) The mean and standard deviation of total points in TFCS-03 and TFCS-04. The maximum value was 100. b) Dropout rates in TFCS-03 and TFCS-04. In TFCS-04, the selected learning style was not explicit, but all students who had at least tried problems, were counted as PBL learners.

Population	Mean	St. dev.
TFCS-03:		
All	58.6	25.0
Trad.	37.4	33.8
PBL	63.3	19.7
Female	64.1	25.2
Male	55.4	24.4
TFCS-04:		
All	60.8	24.3
Trad.	60.5	23.2
PBL	60.9	24.8
Female	66.0	18.9
Male	58.7	25.9

TFCS-03:	
All	15/76=19.7%
Trad.	8/13 =61.5%
PBL	7/63 =11.1%
TFCS-04:	
All	10/56=17.9%
Trad.	3/13=23.1%
PBL	7/43=16.3%

3.2 Exercise points

Because the course evaluation differed in problem-based and traditional ways, we cannot compare the final results. However, the exercise points proved to be a good measure for comparing students in problem-based and traditional ways. The exercise points weighed 25 % of the grade for all students. In addition, we recognize in Table 2a that exercise points and other points (either exam points or sum of problem and learning diary points) have a strong correlation, i.e. exercise points are a good indicator for final results. In Table 2b we have represented the mean and standard deviation values for exercise points. In TFCS-03, the difference is insignificant, but in TFCS-04, the traditional learners collected slightly more exercise points.

Table 2: a) Correlation between the exercise points and other points in TFCS course. b) The mean and standard deviation of exercise points in TFCS-03 and TFCS-04. The maximum value was 60.

Population	correlation coefficient
TFCS-03	
All	0.808
Trad.	0.874
PBL	0.738
TFCS-04:	
All:	0.828
Trad.	0.733
PBL	0.875

Population	Mean	St. dev.
TFCS-03:		
Trad.	37.4	13.6
PBL	38.3	10.1
TFCS-04:		
Trad.	43.6	9.5
PBL	41.8	8.9

3.3 Influence of mathematics

It is quite natural to assume that students' knowledge in mathematics has a strong influence on their success in theory of computation. To test this hypotheses, we evaluated the correlation between the amount of background studies in mathematics (0 cu = none, 1..10 cu = little, >10 cu = much) and the final grade in TFCS course. The results are represented in Table

3. We recognize that there is no correlation or correlation is insignificant when the course is considered as a whole. However, when we consider problem-based and traditional learners separately, we recognize that this holds only for problem-based students, while among traditional learners there is a clear correlation, in TFCS-04 even remarkable. I.e. the students' mathematical background affects on the success, when the course is studied in the traditional way, but in the problem-based way even mathematically inexperienced students can manage well. The reasons may be PBL learner's better motivation to overcome difficulties, and group's support for individual members.

Table 3: Correlation between the amount of studies in mathematics and the grade in TFCS course.

Population	correlation coefficient
TFCS-03	
All	0.043
Trad.	0.242
PBL	0.112
TFCS-04:	
All:	0.258
Trad.	0.806
PBL	0.065

3.4 Female students

It has been reported that the PBL method attracts more female students (Lambrix and Kamkar, 1998). Our experiences (Table 4) are compatible with this hypothesis. In addition, we can recognize in Table 1 that the female students succeeded better in our course. However, we do not have data from other courses to compare average success of female students, whether they are generally better or is it only due to problem-based method. Any way, this is an interesting observation, which should be further studied.

Table 4: Amount of female students in PBL and traditional methods.

TFCS-03	
All	28/76=36.8 %
Trad.	4/13=30.8 %
PBL	24/63=38.1 %
TFCS-04	
All	16/56=28.6 %
Trad.	2/13=15.4 %
PBL	14/43=32.6 %

A natural further question is, if the female students' better success is due to writing skills (learning diary). In Table 5 we recognize that this was not the only reason: the female students got higher points in all categories.

4 Conclusions

In this paper, we have analyzed the effect of problem-based learning in studying one of the hardest courses in computer science curriculum, namely theory of computation.

Table 5: The mean and standard deviation of exercise, problem and learning diary points for female and male students in TFCS-03 and TFCS-04.

	Female mean	Female st. dev.	Male mean	Male st. dev.
TFCS-03:				
Exercises	37.2	13.0	34.1	12.9
Problems	33.7	9.3	32.9	10.1
Diary	17.8	5.8	14.3	6.5
TFCS-04:				
Exercises	39.9	10.3	34.2	16.2
Problems	34.2	7.6	31.1	10.4
Diary	17.1	7.0	14.2	8.3

Our results have proved that students devote themselves better to problem-based learning, their performance is better, and the dropout rate is much lower. The mathematical background affects clearly on success in the traditional way, but in the problem-based way it has no significance. Thus, also poor students manage well in problem-based learning. Our experiences support the known hypotheses that problem-based learning appeals especially to female students, and it could be used to attract more women to computer science discipline.

In this paper, we have not analyzed the students's satisfaction with PBL. However, the analysis of course feedback data has showed that those students who select PBL are satisfied with it, but all students are not eager to adopt a new learning method. Thus, we recommend that students should be able to select the problem-based learning voluntarily.

References

Ben-Ari, M., 2001. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching 20 (1), 45–73.

David, T., Burdett, K., Rangachari, P., 1999. Problem-based learning in medicine. Royal Society of Medicine Press Ltd., Worcester.

Greening, T., Kay, J., Kingston, J., 1997. Results of a PBL trial in first year computer science. In: Proceedings of the Second Australasian Conference on Computer Science Education. ACM, pp. 201–206.

Hakkarainen, K., Lonka, K., Lipponen, L., 1999. Tutkiva oppiminen. WSOY, Porvoo.

Hämäläinen, W., 2003. Problem-based learning of theoretical computer science. In: Proceeding of Kolin Kolistelut – Koli Calling. 3rd Annual Finnish / Baltic Sea Conference on Computer Science Education. pp. 57–65, http://www.cs.joensuu.fi/ whamalai/pbkoli.pdf.

Kirsch, R., 1996. Teaching OLE automation: a problem-based learning approach. SIGSCE Bulletin 28 (1), 68–72.

Lambrix, P., Kamkar, M., 1998. Computer science as an integrated part of engineering education. In: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education. ACM Press, pp. 153–156.

McCracken, M., Waters, R., 1999. Why? When an otherwise successful intervention fails. In: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education. ACM Press, pp. 9–12.

Appendix

Table 6: The seven step model for problem-based learning.

1 Defining unclear concepts:	The students look for concepts, which are unclear and try to define them.
2 Defining problem:	The students discuss about the problem.
3 Brain storming:	Students try to construct, test and compare different hypotheses and explanations.
4 Constructing hypothesis:	The problem is analyzed carefully by comparing different hypotheses. The ideas are argued and organized into an integrated whole.
5 Defining learning goals:	The students write down their learning goals for the self-studying phase.
6 Self-studying:	The students acquaint themselves independently with the topic. In this phase also lectures can be offered to support the self-studying.
7 Sharing the results:	The students compare their solutions and try to help each other to understand the topic. The learning goals are checked and the final conclusions are drawn.

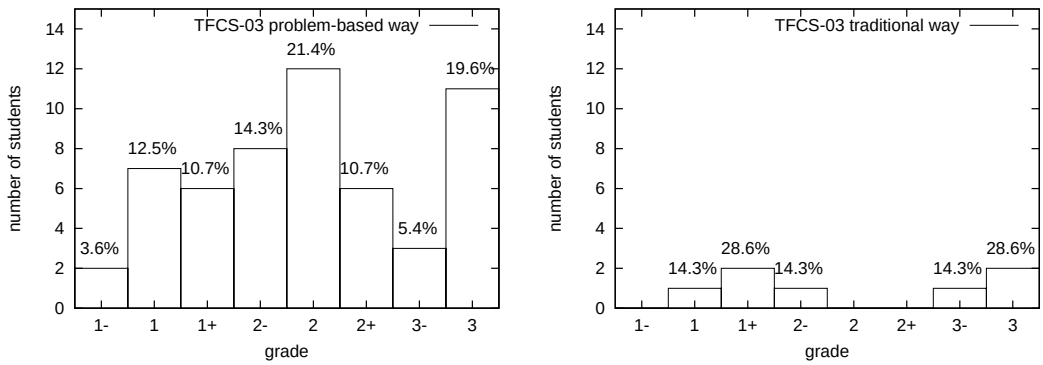


Figure 1: The grade distribution of problem-based and traditional learners in the TFCS-03 course.

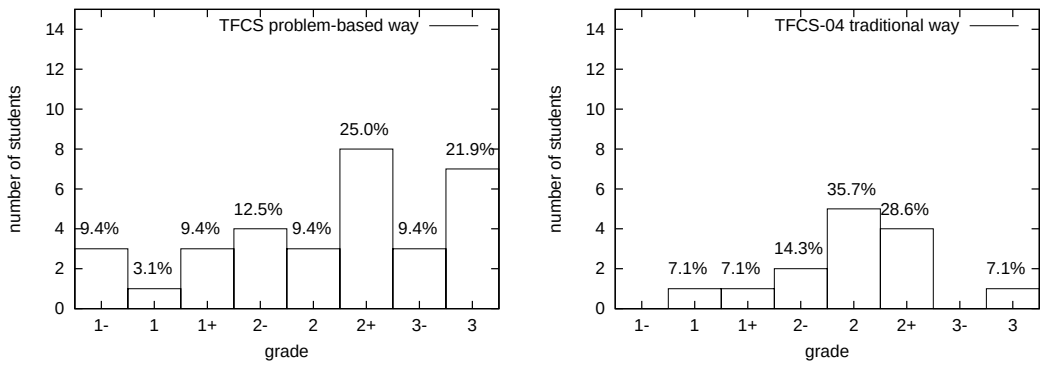


Figure 2: The grade distribution of problem-based and traditional learners in the TFCS-04 course.

Utilizing pedagogical models in web-based CS education

Gaetano La Russa and Pasi Silander
Department of Computer Science, University of Joensuu, Finland

Anni Rytkönen
Department of Computer Science, University of Helsinki, Finland

larussa|psilande@cs.joensuu.fi, Anni.Rytkonen@cs.Helsinki.FI

Abstract

University students have various reasons for failing in their university first year courses. Some solutions and practical implementations have usually been adopted to solve these problems, and in computer science departments attempts have been made to create tailored web-based courses to aid and complement student efforts. This paper intends to introduce new argumentation and suggest some ideas and actions for constructive discussion on how to improve existing systems, and even create new ones. A special focus is set on the use and efficiency of pedagogical models applied to Computer Science education based on web systems and tools. Our scope is also to attempt bridging the communication and methodology gap that exists between lecturers and students.

1 Introduction

Academic studies require students to have a combination of various learning skills and study strategies(Lonka and Ahola, 1995). University first year accessing students have individualized and variegated skills, i.e. of artistic, humanistic or scientific nature, or a combination of them, while each academic field requires specific and defined skills to successfully cope with its given courses. A common problem for students is that they are often unable to see the overall picture of the subjects taught while they are in the process of learning parts of it. In these conditions, the learning can be often described as being accidental and unintentional. In order to avoid this, and to make the learning an intentional and transparent process, teaching should be based on learning processes organized by pedagogical models centered on students' needs. This pedagogical models approach becomes more of a must on web-based courses at the university level.

Students accessing the first year of university computer science (CS) courses typically have differences in their learning methods and know-how of CS. This is partially due to the fact that CS subjects are not taught with standardized contents in high school curriculum (Opetushallitus, 2004) and partially to the fact that students need different learning methods at the university. While in high school, the students are used to be monitored and supervised by their teachers, who keep stimulating students' activities and take care of maintaining their working pace and learning process. At the university, students may find themselves left on their own and without proper monitoring system or support for their learning processes. This is mostly due to the educational system and instructional structures that frame teaching and students' learning culture. In this generalised condition, students tend to give way to disorganized studying, mostly led by external motivation, revealing the learning gaps whenever new complex subjects are presented and taught. In addition, there is the fact that students have to visualize or associate the concepts to be learned, which most of the time are of abstract nature, and learn to use computers as tools for supporting distributed cognition (Karasavvidis et al., 2002).

With the focus on the requirements of CS, this article aims at stimulating a discussion on how to sort out pedagogical models for web-based learning and how to possibly apply these models to the web-based CS education by taking advantage of pedagogical factors. Obviously it has to be kept in mind that the authors' scope is not to set rules or explicitly define the one and best approach to sort and implement pedagogical models, but to stimulate a deeper

analysis on the preconditions and elements that regulate web-based learning. The discussion will gain strength from the existing need to fill the educational gap that lies between the high schools and the university systems. When considering the specific required skills in computes science courses, the overall framework for a web-based learning environment will also be the object of further discussion.

2 Learning environments in computer science

A learning environment can be defined as a combination of all the factors that influence learning (Tella, 1995). It includes the place, the space, the community and its mode of action, and the purpose, of which it facilitates the learning. In an open learning environment (OLE), tutors and learners create contents and maintain the learning process together. When considering the learning process and its contents, an OLE is "open" in the sense that learners can produce new contents, for example exercises (Lifländer, 1989).

Web-based learning environments are OLEs: environments on the World Wide Web enable learners both to build new knowledge and to comment on and discuss it collaboratively. In addition, the knowledge produced by students is often of public domain, in which case the environment forms a common working space for the entire learning community. This may be seen as a corresponding idea to the open source development. An open, web-based learning environment does not necessarily provide the pedagogical structure for activity. The teacher's main responsibility is to construct structures and scaffolds to support learning and student activity in the environment (Koli and Silander, 2003). Teachers' means to facilitate student learning differ according to the various learning theories and the pedagogical knowledge that each singular teacher is aware of and capable to use.

The overall framework for an open learning environment in the context of computer science is presented in Fig. 2. The foundations of proper computer science specific skills and learning behaviour consist of the student's prior learning skills, the pedagogical methods used, and the computer science contents as the subject domain.

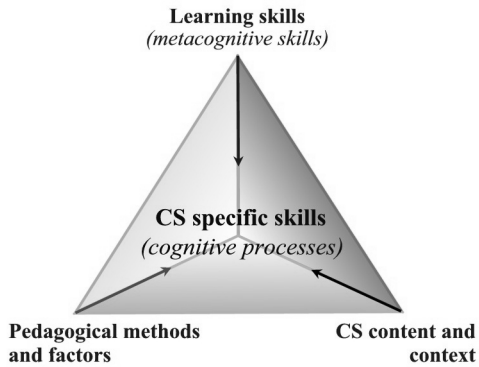


Figure 1: The framework for learning environments in computer science education.

When talking about web-based learning, the educators do not have the same means to influence the students as they have in a lecture hall. Therefore, it is important to design learning situations from the point of view of the learners rather than from that of the teacher. Educators have to use different means to promote and stimulate learning in web-based teaching than they do in traditional teaching. Class teaching is based on the direct teaching process while web-based teaching is based on the learners' learning processes (Koli and Silander, 2003).

3 Factors in pedagogical models that facilitate learning in the web

Learning methods are derived from learning theories, like behaviorism, constructivism or socio-cultural theories, that more or less specify the nature of the learning activity (Gredler, 2001). In order to use web-based learning environments in the pedagogically most efficient way, the learning methods that facilitate the student's learning process should be somehow linked to the use of different features of a learning environment and to the computer science context.

There are no acknowledged pedagogical models that have been created for computer-science-specific learning processes. Therefore, the most optimal way to support a student's learning process is to provide several pedagogical factors in the structure of teaching and the learning task constructed by an educator. The following factors are common to contemporary pedagogical models: progressive inquiry (Hakkarainen et al., 1999, 2001), problem-based learning (Hämäläinen, 2003), activating instruction (Lonka and Ahola, 1995) and Diana-model (Aarnio and Enqvist, 2001) that are used in web-based teaching. The most relevant pedagogical factors are hereafter described:

Learners' prior knowledge and their attitudes toward any subject studied have an influence on learning. New understanding will be founded on prior knowledge. Therefore, it is important that learners first become aware of their prior knowledge.

Problem-oriented learning. Problems arising from the learners' own interest are regarded as a crucial factor in learning. Questions and problems, set by students, direct the knowledge-building process.

Authenticity. The problems to be solved during the learning process should be as authentic as possible. The same applies to assignments: they must not be tasks done just for a teacher. Authenticity requires that the students participate the culture of expertise.

Knowledge-building and the construction of new knowledge, which is meaningful to a learner, have a core position in a learning process. Although learning does not equal knowledge-building, it, along with the formulation of new knowledge, can be conceived as a metaphor for learning.

Externalization of one's own thoughts and the process of problem-solving. When learners externalize their own thoughts and simultaneously the process of knowledge-building, they need to present these ideas in an abstract context. They can then learn from their own externalized reasoning processes and the deficiencies in their thinking become consequently apparent. This externalization creates a need to produce more elaborate constructs.

Collaboration. The meaningful learning is considered to take place in a social context based on collaborative knowledge-building and shared expertise. The externalization of one's own thoughts as well as the feedback from peers plays a significant role here. Social, shared cognition (Karasavvidis et al., 2002) appears in the collaborative knowledge-building process.

4 Learning in the context of computer science

The goals of learning the contents often define the methods needed to be used. If the goal of learning is just remembering or recalling the information, the very basic methods may be used. In this case there is not a great need for students' interactivity or use of sophisticated cognitive heuristics or strategies in the sense of supporting knowledge-building activity.

In computer science, the goal of learning is not just to apply information in various situations. In addition to the factual knowledge and the capability to apply it, the more complex cognitive strategies and processes are needed, e.g. in design of algorithms or data structures. Students should have a capability to constantly create new information and solutions. Therefore more sophisticated mental models and higher levels of cognitive strategies are definitely needed from students. These kind of mental models and cognitive strategies may not be easy to learn just by using traditional methods of teaching that are simply transferred to the web.

The collaborative knowledge-building activity (Bereiter, 2002) and, for example, the symbol systems as tools for thinking are needed.

Due to the highly abstract contents of computer science, students are required to have a variety of skills. In order to learn the skills needed to apply knowledge in various situations, the information should be deeply processed by a student. There are needs for interactivity that enable reflection and learning-by-doing. There may also be scaffolds as triggers for the student's cognitive processes and elements that methodically support the student's information processing like problem solving. A web-based learning environment may provide cognitive tools or scaffolds for the problem-solving and knowledge-building process.

5 Discussion

Learning in CS should be a qualitative and conceptual change in a learner's comprehensive knowledge structures and cognitive processes. The aim of e.g. information processing and problem solving in the web-based learning process is to create conceptual change, i.e. genuine qualitative change in the student's knowledge structures and processes. Just adding new information to existing knowledge is not sufficient from the point of view of learning computer science. A conceptual change is often a prerequisite for the capability to apply the substance learned in a different situation and in practice as well as prerequisite to create new solutions. The most prevailing challenge of web-based computer science education at the university is to provide insightful learning processes for students through applied pedagogical models in a computer science context.

References

Aarnio, H., Enqvist, J., 2001. Dialogic knowledge construction as the crucial issue in network-based learning in vocational education. Proceedings of the 13th World Conference On Educational Multimedia, Hypermedia and Telecommunications , 1–6.

Bereiter, C., 2002. Education and mind in the knowledge age. New Erlbaum Associates, Hillsade, NJ.

Gredler, M., 2001. Learning and instruction - Theory into practice. Merill Prentice Hall, NJ.

Hakkarainen, K., Lonka, K., Lipponen, L., 1999. Tutkiva oppiminen - älykkään toiminnan rajat ja niiden ylittäminen. WSOY, Porvoo.

Hakkarainen, K., Rahikainen, M., Lakkala, M., Lipponen, L., 2001. Perspectives of CSCL in Europe: A review. A report for the European Commission, ITCOLE Project , 30–45.

Hämäläinen, W., 2003. Problem-based learning of theoretical computer science. Proceedings of the 3rd Annual Baltic Sea Conference on Computer Science Education , 57–65`http://www.cs.joensuu.fi/~whamalai/pbkoli.pdf` [24.9.2004].

Karasavvidis, I., Kommers, P., Stoyannova, N., 2002. Preface: Distributed cognition and educational practice. Journal of Interactive learning research 13 (1), 5–9.

Koli, H., Silander, P., 2003. Web based learning - designing and guiding an effective learning process. Häme Polytechnic, Hämeenlinna.

Lifländer, V.-P., 1989. Tietokoneavusteisen opetuksen kehittäminen. Tech. Rep. D-112, Helsinki School of Economics.

Lonka, K., Ahola, K., 1995. Activating instruction - how to foster study and thinking skills in higher education. European journal of psychology of education 10 (4), 351–368.

Opetushallitus, 2004. Lukion opetussuunnitelman perusteet 2003. http://www.edu.fi/julkaisut/maaraykset/ops/lops_uusi.pdf [24.9.2004].

Tella, S., 1995. Virtual School in a Networking Learning Environment. OLE Publications 1. Department of Teacher Education, University of Helsinki, http://www.helsinki.fi/people/seppo.tella/ole1.html [24.9.2004].

Where Have All the Flowers Gone? — Computer Science Education in General Upper Secondary Schools

Tanja Kavander and Tapio Salakoski
Department of Information Technology
University of Turku and Turku Centre for Computer Science

tanja.kavander@it.utu.fi

Abstract

The status of computer science education in Finnish general upper secondary schools ("high schools") has changed. Computer science was earlier regarded as a basic skill containing different kinds of special areas and thus a meaningful part of all general education; in fact, computer science was included in the national core curriculum as an independent subject. Recently the status has changed as computer science is presented only as a tool for other subject studies. In this paper, we examine the cultural change in computer science education by analyzing the national core curricula of general upper secondary schools in Finland from 1972 to 2003.

1 Introduction

As a science matures, it can be taught at lower levels. According to Ben-Ari (2004) this has now also become the case for Computer Science (CS), although it is a rather young field. Graduate research topics can be found on undergraduate courses and within a few years these things can be taught in high schools. However, in the Finnish General Upper Secondary (GUS) education ("high schools") this does not seem to be true; when examining the official National Core Curricula (NCCs) in recent years it becomes obvious that teaching CS as an independent subject is no longer considered essential. Clearly, the focus has changed.

Today there are many different approaches to teaching CS in GUS. Gal-Ezer and Harel (1998) differentiate between three ways of instruction: the true CS education being confused with disseminating computer literacy or with the use of computers in teaching other subjects as computers are integrated into instruction in a variety of ways.

We start by examining the history of CS education in the GUS education by comparing the structures of different NCCs. Thereafter the changing history of CS is examined in more detail as well as the number of CS references in other subject areas. A short survey of CS related studies in other compulsory minimum curricula of European countries is presented next. Results and indications of matters investigated are discussed in the concluding section.

2 The NCCs for GUS Schools

The main purpose of the NCC is to serve as a base for the planning and implementation of instruction in the GUS education. It has been common for a new NCC to be introduced once every decade.

NCCs were still delivered as paper documents to the heads of the schools in 1972. The documents contained one single syllabus for each subject, giving descriptions on how a given subject was to be taught during a whole school year; back then, instruction in a specific subject was not divided into separate courses. For instance, the document for mathematics consisted of one advanced and one basic syllabus, also called long and short courses in mathematics (Kouluhallitus, 1972). The document started with an introduction to the learning of mathematics, followed by a chronological listing of the contents for every school year. Thereafter the optional studies were presented along with a more detailed description of the studies. The same structure was also used in the following mathematics curriculum (Opetushallitus, 1973).

In the curricula of 1994 and 2003 the importance of the school-specific curricula were emphasized (Opetushallitus, 1994, 2003). The fulfillment of the schools' aims at teaching different subjects were enhanced with cross-curricular themes, whose purpose was to rearrange the teaching of separated subjects more into a unity of studies. Themes were already introduced in NCC in 1985, then without any specific content, only stating that by choosing the themes, subject areas like society, economy, nature, and culture were made to be more uniform (Kouluhallitus, 1985). In the newest NCC, themes are listed explicitly also giving specific goals and descriptions for them (Opetushallitus, 2003).

There is also another observable change in NCCs of the GUS education. Lately more and more power has been moved to the lower hierarchies in the education system, giving local schools more power in deciding on how the curricula are to be implemented. The earlier curricula (Kouluhallitus, 1972; Opetushallitus, 1973) contained orders presented by the government on which subjects were to be taught at schools and what the instruction should contain. Starting from NCC of 1985, only the main contents and objects of different studies have been presented (Kouluhallitus, 1985), leaving the decision on how the subjects should be presented and taught to instances at municipality level. In the newest NCC the importance of implementing different curricula in different schools in the same municipality is even emphasized (Opetushallitus, 2003). At the same time the importance of a single teacher as a producer of the curriculum has increased.

3 The Changing Position of CS in GUS Education

In this section we examine the vacillating status of CS education in the NCCs in more detail. Basic knowledge of CS, earlier known as Automated Data Processing (ADP), was mentioned for the first time in 1970 in a letter sent by the National Board of Education (Opetusministeriö, 1970). The letter listed special courses to be included into the mathematics syllabus, and CS was mentioned along with Boolean algebra.

The NCC presented in 1972 contained a special course of ADP included in the advanced syllabus of mathematics. There was no description for the course whereas the basic syllabus contained a more detailed description of the course of ADP: the phases of data processing, most important terms and possibly some demonstrations. Excursions were mentioned as a way to substitute the demonstrations (Kouluhallitus, 1972). The same kind of guidelines can also be found in the following curriculum (Opetushallitus, 1973).

The syllabi list in the NCC of 1985 contained a description of CS studies, then called Information Technology (Kouluhallitus, 1985). Two CS courses were described: one dealing with the basic use of computers and the second entirely devoted to programming. The courses were marked as optional courses and the school was obliged to offer at least one course of CS for the students (Kouluhallitus, 1985). The curriculum contained detailed lists of topics that should be included in a specific syllabus. Each syllabus was described using the term subject, however, the differences are noticeable when comparing the references to compulsory and auxiliary studies. In the NCC of 1994 this problem was resolved by defining that each course mentioned in the curriculum defines a subject (Opetushallitus, 1994). It is worth mentioning that without explanations CS was no longer mentioned as a subject in this NCC. Instead, the cross-curricular themes, received a more vital position in the curriculum, can now be seen as a reflection of CS.

In NCC 2003 themes that mainly cover CS are "technology and society", and "communication and media competence"; two of the six themes mentioned altogether (Opetushallitus, 2003). Technology is defined as the development of knowledge and skills needed in technology design and production and use of technological products, processes and systems. Students are to be guided to understand, use and master technology according to the aims of the theme. Innovation and problem solving skills are also emphasized. In the media related theme students' ability to produce, co-operate and cope with media is accentuated. Neither of these themes contains any actual references to computers or CS.

Table 1: CS related references from different curricula grouped by the context in which they are mentioned.

Curriculum	Information Management	Computers and CS	Subject Specific
1985	1	4	2
1994	6	6	4
2003	5	5	12

Table 2: CS related references from different curricula grouped by the place where the references are situated.

Curriculum	Introduction	Goals of the Syllabi	Course Descriptions
1985	1	2	4
1994	1	10	5
2003	3	8	11

4 How is CS Referred to in Other Subject Areas?

In this section we examine references made to CS or closely related topics in the three newest curricula, which all are in the format of a book containing all the syllabi for the curriculum. We believe the number of these references reflects the significance of the information technology in the society. We also think that the changing position of CS can be observed studying the nature of the references.

In the analysis we have omitted references that can be regarded as something else than a true CS related topic. Various references to concepts such as "media", "digital and electronic information" and "technology" are also ignored. Instead we have accepted references like "algorithms", "programs and databases" and "the application of new technology in information processing".

Table 1 lists the references grouped by the context in which they are mentioned. The references have been divided into the following categories: the usage of CS for managing information; computers or CS seen as an independent area which is of interest to the subject area; and references stating a well-defined subject specific role for CS. It can be seen, that in all categories the number of references tends to be increasing.

In the table 2, the same references are categorized according to the place where they occur in the curricula. "Introduction" stands for the start for every curriculum, giving an introductory presentation of the aims and goals of the work at school. "Goals of the syllabi" and "Course Descriptions" are references in the descriptions of subjects accordingly. The distribution of references has been steady, but in the newest curriculum the amount of references in course descriptions has increased.

Of course, these numbers can not be truly directly compared since there have been some changes between the different NCCs; for instance, the number of subjects included in the curricula have varied. In addition, the style in which the contents of the courses has also differed from one curriculum to another. Furthermore, the use of themes has made it possible to refer to a theme instead of having a direct reference to CS. However, preliminary results suggest further studies.

5 The Status of CS Studies in Europe

Studies of information and communication technology (ICT) forms a part of the compulsory curriculum in the upper secondary level in all European countries except for Italy, according

to a comprehensive study of ICT studies made by Eurydice (2004), an EU-based organization consisting of the 25 EU-member states, and Bulgaria, Iceland, Liechtenstein, Norway, and Rumania. The two approaches of teaching ICT, as a separate subject and as a tool for other subjects, are combined in most cases. Most curricula also recommend the teaching of ICT as a subject to be supplemented with its use for introducing other subjects or carrying out interdisciplinary projects. However, Finland and Ireland are the only two countries, except for Italy, where ICT is described only as a tool in curriculum of GUS education, elsewhere also the subject of ICT is defined. (Eurydice, 2004).

6 Conclusion

In the short history CS was first included in mathematics. The NCC of 1984 stated that CS was to be taught as an individual subject, but later on there have not been any requirements for separate instruction in CS; instead CS has been integrated in cross-curricular themes. When considering the references to CS in other subject areas, the tendency seems to be increasing. However, even if the status of CS is increasing in the Finnish society, for example by the growing number of industry related to the area, the actual body of knowledge of CS in the GUS education is still being defined only as a tool in other subjects. The rapid development of the CS research is being kept in disguise. Nevertheless, more research is needed before further conclusions regarding the changes to CS education and its status in relation to other subject areas can be made.

CS education is at the hands of each single GUS school since each municipality has the decision power and CS is taught in applied courses that schools do not have to arrange and students do not have to take. The local curricula are developed at municipality level making the development of CS education dependent on individual teachers to actively improve and teach courses dedicated to CS. Enthusiastic teachers should therefore be cherished as precious assets, as they are the key actors in the future of CS education at the GUS level; these teachers decide whether courses in CS are arranged or not. Shortly put, the possibility to learn CS in Finnish high schools is changing to a game of chance.

References

Ben-Ari, M., 2004. Computer Science Education in High School. Computer Science Education 14 (1), 1–2.

Eurydice, 2004. Key Data Information and Communication Technology in Schools in Europe. Eurydice, The information network on education in Europe, Belgium.

Gal-Ezer, J., Harel, D., 1998. What (Else) Should CS Educators Know? Communications of the ACM 41 (9), 77–84.

Kouluhallitus, 1972. Lukion matematiikan opetussuunnitelmat.

Kouluhallitus, 1985. Lukion opetussuunnitelman perusteet. Valtion painatuskeskus, Helsinki.

Opetushallitus, 1973. Lukion matematiikan opetussuunnitelmat.

Opetushallitus, 1994. Lukion opetussuunnitelman perusteet. Valtion painatuskeskus, Helsinki.

Opetushallitus, 2003. Lukion opetussuunnitelman perusteet 2003. Nuorille tarkoitetun lukiokoulutuksen opetussuunnitelman perusteet. Vammalan Kirjapaino Oy, Vammala.

Opetusministeriö, 1970. Matematiikan erikoiskurssin oppiennätysten vahvistaminen.

Strict Logical Notation Is Not a Part of the Problem but a Part of the Solution for Teaching High-School Mathematics

Mia Peltomäki and Tapio Salakoski
Department of Information Technology
University of Turku and Turku Centre for Computer Science

mia.peltomaki@utu.fi

Abstract

During the years 2001-2004 we have conducted an experiment at a Finnish high school, where we taught advanced mathematics to students interested in computer science using a new teaching method based on structured derivations and their computer-assisted use. With the new method, we aim at a deeper abstract logical thinking via more exact and perceptible expression. In this paper, we describe the experiment and present the preliminary results. The quantitative learning results show that in general, the new method is definitely not worse than the traditional way and for talented students it yields substantially better results. Furthermore, the method clearly has the potential of improving the quality of learning.

1 Introduction

At high school and lower level education, using exact formalism in definitions and proofs is often avoided even in mathematics courses and thus the design of proofs is not taught either. Students do see a few proofs and may even be asked to develop a few themselves, but usually there is only a little or no discussion at all about the principles or strategies for designing proofs. The concept of mathematical proof, or computational derivation, is not made perceptible to the student and, consequently, students ability of logical reasoning and abstract thinking remain poor. It seems that high school does not give students the sufficient thinking tools for employing mathematics in solving practical problems, a skill especially needed at university level computer science education.

After nine year-long basic education in Finland, students aged 15 or 16 continue mainly either in general upper secondary school or in vocational upper secondary education and training. In this text, we use the phrase high school for Finnish general upper secondary school, which is public and free for all students. The teaching is given at courses consisting of about 30 classroom hours. After completing at least 75 courses students aged 18 or 19 participate in the national matriculation examinations.

According to the current high school level curriculum, computer science is no more mentioned as an independent subject. The aims of mathematics studies include among other things that students appreciate the exactness and clarity in presentations. The new curriculum to be adopted in 2005 emphasizes the evaluation of students skills focusing to the selection of an appropriate method, and the justification of exact conclusions. Learning to see the mathematical information as a logical structure has been added to the teaching aims. (Opetushallitus, 1994, 2003)

Despite the noble objectives, logical notation is in practise very seldom used at high school. Logic, quantifiers, and the training of proving principles come along with the new curriculum, but only on the optional advanced courses following the compulsory courses. We believe, however, that it would be better to explicitly use logic already on the compulsory courses – but as a tool rather than an object of study.

Starting with logic makes the course seem coherent and provides students with a supportive framework, which they can lean on while the various aspects of the proof and counterexample are falling into place. It builds students confidence in the rationality of the mathematical enterprise and helps allay their fear of failure. Determining truth and falsity of mathematical

statements is so complex, that even when they are motivated, students often fail to really "get it" if they do not have any prior experience with basic logical tools. (Epp, 2003)

In this paper, we show how structured derivation, which is an extension on the calculational proof style written with logical notation, could be used in high-school mathematics to facilitate the students to write the justifications of the solutions. Our objectives are the following. Firstly, we want to promote exact expression by employing formal notation. Secondly, we try to make mathematical thinking and logical reasoning visible and hence more perceptible. Finally, we wish to deepen the students understanding of logical abstractions and the processes of thought by making use of computer-assisted manipulation of structural derivations.

The paper is organized as follows. In Chapter 2, we very shortly introduce structured derivations. Our teaching experiment at is described in Chapter 3, and in Chapter 4, we present preliminary results. We end the paper by discussion and conclusions.

2 Structured Derivations, briefly

Dijkstra and Scholten (1990) introduce the calculational proof format in their book *Predicate Calculus and Program Semantics*. They begin by making the observation that a great many proofs can be described as a series of transformations. Inspired by the clarity and the readability of the format, calculational paradigm for manipulating mathematical expressions emerged. According to the paradigm, mathematical expressions are transformed step by step from the initial expression to a solution. Each new version of the expression is written on a new line and between the two lines is written a symbol denoting the relationship between the expressions together with a justification for the validity of the step. The paradigm has been attributed to W. Feijen, and described in detail by van Gasteren (1990).

As an example of applying the method on the courses in detail, the derivation of solving an inequality is shown both in its fully expanded form as written on the whiteboard and in the form with hidden sub-derivations (Example 1).

Example 1.
Solving the inequality $|x-4| \geq 3x$ shown
(a) in the form with hidden sub-derivations and
(b) in its fully expanded form as written on the board

(a)

$|x-4| \geq 3x$
$\equiv$ {property of absolute values}
$(x-4 \geq 3x \vee x-4 \leq -3x)$
$\equiv$ {solve the inequalities}
$\ldots\ x \leq -2 \vee x \leq 1$
$\equiv$ {simplify}
$x \leq 1$

(b)

$|x-4| \geq 3x$
$\equiv$ {property of absolute values}
$(x-4 \geq 3x \vee x-4 \leq -3x)$
$\equiv$ {solve the inequalities}
- $x-4 \geq 3x$
  - $\equiv$ {add 4 to both sides of the inequality}
    $x \geq 3x+4$
  - $\equiv$ {subtract $3x$ from both sides of the inequality}
    $-2x \geq 4$
  - $\equiv$ {divide both sides with -2}
    $x \leq -2$
- $x-4 \leq -3x$
  - $\equiv$ {add 4 to both sides of the inequality}
    $x \leq -3x+4$
  - $\equiv$ {add $3x$ to both sides of the inequality}
    $4x \leq +4$
  - $\equiv$ {divide both sides with 4}
    $x \leq 1$

$\ldots\ x \leq -2 \vee x \leq 1$
$\equiv$ {simplify}
$x \leq 1$

Although structured derivations are based on the calculational proof paradigm and they are developed originally for the formal refinement of computer programs and reasoning about their correctness (Back and von Wright, 1998) this method can be used in high-school math-

ematics as a way of writing solutions to typical problems (Back and von Wright, 1999; Back et al., 2002). Furthermore, when solutions are very long, sub-derivations can be hidden and replaced with a link giving more detailed view of the partial solution.

3 Teaching experiment

The method has been studied and tested at a Finnish high school between August 2001 and May 2004 (Kavander et al., 2001; Back et al., 2002, 2003). Every year before beginning the studies, the new students in advanced mathematics answered to an attitude enquiry. They also took an exam, which tells us about their mathematical skills. The students were then divided into three groups: the test group, the control group and the rest.

The test group was selected so that most students expressing their interest in computer science were included, because we believe that the new method would turn out especially useful later when studying computer science at university level. Nevertheless, we tried to divide students to the test and control groups so that they were about on the same entry level according to the basic education certificate and the mathematical skills tested in our exam. Randomizing would certainly have been the best way to form the test and control groups; however, in the Finnish high school system, the students choose their classes themselves. Thus in practice, it is impossible to establish randomized groups which could then be maintained for three years during the ten compulsory mathematics courses.

We used the structured derivations method when teaching new basic theory both on the whiteboard and with a computer. When using a computer and a browser, sub-derivations, consisting of a detailed solution to a part of the problem can be hidden with a link or shown depending, on how detailed a solution the students want to see for instance when checking their homework.

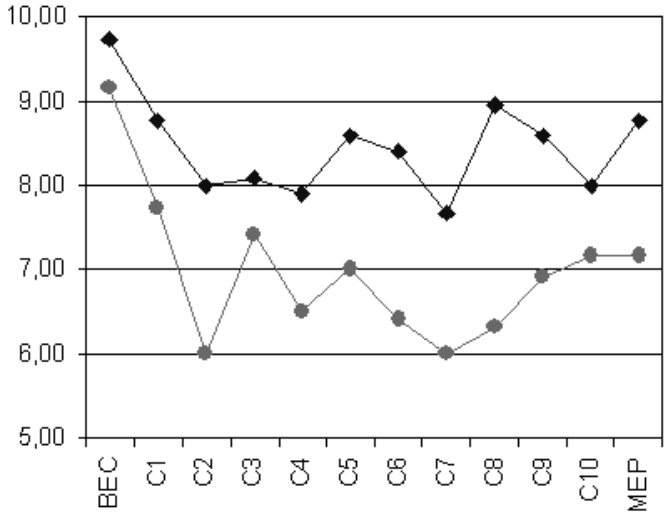
The results of the test group have been compared with those of the control group throughout the three-year period by arranging common examinations for each course. Finally, after finishing high school, we compared the results of the matriculation examination between the students in the test group, the control group, and all students in Finland participating in the national examination.

4 Results

When comprehensive school students enter high school, it is customary that the grades fall. However, it seems that the group participating in the teaching experiment unconsciously received a slightly stricter course assessment than usually; most likely due to the high entry level. It can be seen that the mark 10 typically falls down to about 8.5 (Suikki, 2004) whereas in the test group, it fell down to 8.2. Consequently, the results of the matriculation examination were better than expected based on the course assessments. The accepted course grades range from 5 to 10 and here, we have scaled the grades in the matriculation examination respectively.

A more detailed analysis of the performance of the test group in the national matriculation examination is still going on. Preliminary results suggest, however, that the test group succeeded remarkably well. More than 70% of the test group got one of the two highest degrees (here, 9 or 10) while on the national level this is normalized to about 20% (in spring 2004, 21%). Detailed results are not shown here.

In Figure 1, we present a summary of the preliminary results. We consider the development of the grades from the basic education certificate, through the ten compulsory courses of advanced mathematics, to the matriculation examination results.



	test	control
Basic education certificate (BEC)	9,73	9,17
Mean of 10 compulsory courses (C1-C10)	8,30	6,75
Mean on matriculation examination points (MEP)	8,77*	7,17

*One student quit mathematics and dropped out. Counting this as a failure (zero points) would result the average 8,52.

Figure 1: The grades results. The means of the mathematics grades in the basic education certificate (first in left); the means of the course grades of the ten compulsory courses (in the middle); and the means of the matriculation examination points (last in right) in the test and control groups in a scale from 5 to 10.

5 Discussion and conclusions

By using the logical notation and the method of structured derivations, as suggested in this paper, logical reasoning and abstract thinking can be made more perceptiple to the students. The logical structure of the stepwise derivation from the original formulation of the problem to the final solution, is maintained while the level of abstraction can be adjusted depending on the focus of consideration.

The quantitative learning results of the teaching experiment show that in general, the use of logical notation and structured derivations in high-school mathematics is certainly not less productive than the traditional way and, moreover, the new method seems to yield substantially better learning results for talented students. In addition, although a thorough qualitative analysis of the solutions by the test group students is still under way, the new method seems to improve the quality of learning, resulting in deeper understanding of logical abstractions and advanced mathematical thinking.

Based on our experience, most of the students are deeply committed to the style they have learned before high school. The standard textbooks used on high-school mathematics courses do not include logic. As the teaching progressed with the experimental method using structured derivations and not according to the textbook, some students got mixed and consequently, their results may have suffered. Obviously, this problem of inconsistent learning material could be avoided, if a suitable textbook existed to support the use of the new method.

Gries and Schneider (1993) have used the calculational proof style in their book *A Logical Approach to Discrete Math*, and their experiences on teaching in this way at the university level have been positive (Gries and Schneider, 1995). Our conclusion at the high-school level is similar: talented students learned easily the basic logical tools and also, they seem to make better use of the new formalism, whereas some of the less-talented students got confused with the new and traditional notation and thus their solutions to the exercises were not always logically firm. Therefore, we think that at the high-school level, the notation should be somewhat lighter than at the university level. During this three years experiment we have already revised the notation accordingly, and placed a section about the basics of logic into the beginning of the first course.

We look forward to preparing teaching material for the courses, perhaps in the form of a

textbook. In addition, we aim at developing tools for writing structured derivations quickly and easily with a computer. These tools would enhance the usability of the method from the students as well as from the teachers point of view, and they could be accompanied by various guidance, verification, and assessment features. Furthermore, we are currently revising the method for incorporating a systematic way of using pictures and naming conventions for problem solving, such as in experimental settings, geometrical illustrations and data structures.

References

Back, R., Kavander, T., Nylund, M., Peltomäki, M., Salakoski, T., von Wright, J., 2002. Matemaattinen todistaminen ja abstrahointi lukion matematiikan opetuksessa – opetuskokeilu rakenteisten johtojen avulla. (Mathematical Proof and Abstraction in High-School Mathematics Education – Teaching Experiment Using Structured Derivations). In: Meisalo, V. (Ed.), Aineenopettajankoulutuksen vaihtoehdot ja tutkimus. Ainedidaktiikan symposiumi 1.2.2002. Tutkimuksia 241. Department of Teacher Education, University of Helsinki, pp. 691–697.

Back, R., Peltomäki, M., Salakoski, T., von Wright, J., 2003. Structured Derivations Supporting High-School Mathematics. In: Proceedings of 20th Annual Symposium of the Finnish Mathematics and Science Education Research Association. Department of Teacher Education, University of Helsinki, to appear.

Back, R., von Wright, J., 1998. Refinement Calculus: A Systematic Introduction. Springer-Verlag, New York.

Back, R., von Wright, J., Mar. 1999. Structured Derivations: a Method for Doing High-School Mathematics Carefully. Tech. Rep. 246, TUCS - Turku Centre for Computer Science.

Dijkstra, E., Scholten, C., 1990. Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science. Springer-Verlag, New York.

Epp, S., 2003. The Role of Logic in Teaching Proof. American Mathematical Monthly (10), 886–899.

Gries, D., Schneider, F., 1993. A logical Approach to Discrete Math. Texts and Monographs in Computer Science. Springer-Verlag, New York.

Gries, D., Schneider, F., 1995. Teaching Math More Effectively Through the Design of Calculational Proofs. American Mathematical Monthly , 691–697.

Kavander, T., Nylund, M., Peltomäki, M., Salakoski, T., von Wright, J., 2001. Teaching High-School Mathematics with Structured Derivations in Hypertext Format. In: Proceedings of Kolin kolistelut – Koli Calling. The First Annual Finnish/Baltic Sea Conference on Computer Science Education. Report series A. University of Joensuu. Department of Computer Science, pp. 46–52.

Opetushallitus, 1994. Lukion opetussuunnitelman perusteet. (Principles of the High School Curriculum, in Finnish). Valtion painatuskeskus, Helsinki.

Opetushallitus, 2003. Lukion opetussuunnitelman perusteet. (Principles of the High School Curriculum, in Finnish). Valtion painatuskeskus, Helsinki.

Suikki, M., 2004. Arvosanojen korrelaatiotutkimusta. (Correlation studies on comprehensive and high-school grades). Dimensio (4), 16–21.

van Gasteren, A., 1990. On the Shape of Mathematical Arguments. Lecture Notes in Computer Science. Springer-Verlag, Berlin.

Survival of students with different learning preferences

Roman Bednarik and Pasi Fränti
Department of Computer Science, University of Joensuu, Finland

roman.bednarik@cs.joensuu.fi, pasi.franti@cs.joensuu.fi

Abstract

We study learning preferences of CS students and academics and their correlations to the learning results in three advanced courses. We measure visual, aural, read/write and kinesthetic perceptual modalities and relate them to the results. We classify the students first into four most distinct groups and second into the groups according to the highest preference. The results suggest that although in average the groups perform similarly, for a certain course some groups perform better. The findings are also correlated with the perceptual modalities of teachers, suggesting that a group with a modality distribution close to the teachers' might perform better than a more distant group.

1 Introduction

Individuals differ in the way they learn. As teachers, we often realize that we have little or no knowledge about the learning styles of the students of our courses and therefore we cannot customize the way we deliver the courses to be in an appropriate format. The customization and attempts to identify the individual differences of learners and personalize the instruction often end up in, for instance, mere adjustment of the subject matter taught to meet the curricular requirements and prior knowledge level of students. Indeed, also Gardner et al. (1995, p. 254), although in the context of intelligence, recognizes the *"considerable importance"* of the issue *who does well in school*. In order to fill this gap of knowledge, we performed a study using the VARK learning preference assessment (Fleming, 2001) and compared the performance of students to their learning preferences. We also studied the correlations between the learning preferences of the students and the profiles of their teachers.

Learners perceive and process the information in several modalities. The learning style preference indicates how usually a student learns, processes and integrates information in different situations. Learning style preference, however, changes over the time. With gained experience some preference can be masked with other or even suppressed. Therefore, the preferred perceptual modality as measured at a time cannot lead to a final conclusion about an individual in learning process. It has to be kept in mind that a learning style preference is rather a simplified concept and does not describe a learner wholly. Moreover, the VARK profile does not describe other abilities thought to be foundational for a successful learning. Another methods to estimate the learning preferences have to be used, for instance those for the group vs. individual work preference, for abstract vs. concrete, for active vs. reflective perception, etc. Considering the input and output of information, it is believed that a certain modality preference for input is also reflected in the same preference for the output, and vice versa. The results shall be interpreted as a recommendation only, and not as a complete assessment. The preferred modality shall indicate the way of taking and producing the information to achieve a maximum learning effect. Similarly, the results can reveal the possible difficulties if a certain learning strategy is imposed on a student.

Several assessment instruments of learning style preferences exist. For the present research, we used the VARK method and questionnaire (Fleming, 2001). The acronym VARK comes from the four modalities that are assessed from the results of the questionnaire: ***V****isual (sight),* ***A****ural (hearing),* ***R****ead/****W****rite,* and ***K****inesthetic*. A learner showing a priority in the Visual modality prefers learning using charts, symbolic representations, or video materials, over the auditory input as e.g. instructor's explanations or verbal presentations. The Aural/auditory perceptual preference indicates that a learner perceives best the information, which is auditory, verbal and therefore heard. Aural students prefer discussions, chats, explanations in

words, and lectures. The R/W modality refers to the input and output through reading and writing, representing the information in text rather than in sound or graphics. According to (Fleming, 2001), it is found strongest in most of the academics. The Kinesthetic preference emphasizes the hands-on experience over the other modalities, stressing the importance of real or simulated practice.

2 Experiments

2.1 Courses and participants

The experiment was conducted on three advance-level courses held during the academic year 2003-2004, at the Department of Computer Science, University of Joensuu, Finland. The VARK questionnaire was first implemented in the DAA course, and most of the students answering the questionnaire have results for this course. The questionnaire was then administered in the AI course. Later, included also the IMA course into the study, as many of the students had taken this course, too. The first author was one of the assistants in the AI course, and the second author was the lecturer in the DAA and IMA courses.

The DAA course (Fränti, 14.6.2004a) is a 14 weeks, 4 credit-units (cu) advanced course, covering the concepts and principles of algorithms, their implementation, analysis and practical use. The course was delivered in a lecture-demonstrations-exams format, scheduled every week with the two two-hour lectures and one two-hour demonstration session. Students pass the course by performing two exams, visiting demonstrations and completing the assignments.

The 3 cu AI course (LaRussa, 14.6.2004) was delivered in an interactive format: a week before a lecture (scheduled two hours per week, totally 15 weeks), the lecturer announced the topic and a relevant literature to students, who studied the subject matter in advance. The lecture time was dedicated for a discussion about the present topics rather than for lecturing. The students kept a learning diary and processed selected topics off-class, delivering both of them weekly. The topics of the independent homework were discussed in a weekly held two-hour demonstration session. The AI course project formed 60% of the final grade.

The 3 cu IMA course (Fränti, 14.6.2004b) discusses the topics of image processing and analysis, such as histogram manipulation, shape and motion detection, or image retrieval. The course is scheduled in 10 weeks, with 2 two-hour lectures and one two-hour exercise session per week. The requirements for passing the IMA course were about similar as those of DAA course.

The participants were Finnish undergraduate and graduate students, international graduate students mainly from Russia (but also from Romania, India, Czech Republic, Spain, and El Salvador). All together there were 42 student (14 females) profiles analyzed, from which 19 were from foreign students. We collected 32 profiles from DAA, 12 from AI, and 17 from IMA course. We also collected the profiles of the involved teachers.

2.2 VARK questionnaire

The VARK questionnaire consists of 13 multiple-answer questions presenting everyday situations and providing three or four answers related to one of each VARK categories. The subject can mark one or more answers to the problem. Marking an answer gives a point for the particular category. In total, answering all questions and marking all possible preferences yields into 12 points for each of V, A, R, and K. To enforce some level of validity, at least 10 questions answered are required. A sample question in the questionnaire is below.

> You are about to give directions to a person who is standing with you. She is staying in a hotel in town and wants to visit your house later. She has a rental car. I would:
>
> - draw a map on paper. (**V**)
> - tell her the directions. (**A**)
> - write down the directions (without a map). (**R/W**)
> - collect her from the hotel in a car. (**K**)

An on-line VARK questionnaire was used, allowing for an easy maintenance and accessibility, and for an automatic collection of the results through email. Apart from VARK scores, the results also contained the highest modality and its strength, which were computed as a highest score among the modalities and its distance from the second highest, respectively. The scores were normalized by diving by the total number of answer of the person in order to allow for a comparison between subjects.

Our hypotheses were following: 1. Certain learning types perform in general better than other types. 2. Students with a similar VARK profile to teachers might perform better.

We suggest that the teacher and his/her way of delivering a course interplay with the students of certain learning preference. For example, if the teacher is of the visual type, s/he unconsciously provides the information best in the visual format. Therefore, the students who are of visual type, might understand the presented knowledge better than others.

3 Results and Discussion

3.1 VARK profiles of participants

Table 1 presents the profiles of academics in the study. The teacher and assistant of the DAA and IMA courses show a strongest preference in the Visual modality. The teacher of the AI tends to prefer the aural perception, while the assistant in the AI course shows a balanced preference between Visual and Kinesthetic modalities. Overall, the average profile of the academics in this study showed 31% of Visual, 22% of Aural, 20% of Read/Write, and 28% of Kinesthetic preference.

Figure 1 shows the profiles of all the students. The average normalized VARK profile computed from all the profiles shows a distribution of 23%, 21%, 24%, and 31% of V, A, R, and K, respectively. The modalities were varying as follows (min-max %): Visual 6-45%, Aural 0-43%, Read/Write 6-44%, and Kinesthetic 15-50%. It can be observed that there were students without any Aural preference, while everybody had at least 15% of Kinesthetic preference. In order to investigate whether there was a relation between the VARK variables, we computed their correlation. No significant correlation between VARK variables was observed, which indicates that they were independent on each other.

3.2 Classification and analysis of students' profiles

We investigated the hypothesis that there might exist four distinct clusters, corresponding with the four prevailing V, A, R, K modalities within a cluster. We grouped students into four clusters according to the VARK profiles in four dimensional VARK Euclidean space, using the algorithm as in (Fränti and Kivijärvi, 2000). The similarity within a cluster was maximized. Figure 2 shows four groups into which the students were divided, also shown as vertical lines in Figure 1. Group 1 profile is characterized by its low preference in visual modality, but relatively evenness of all others. Group 2 has a profile with high visual and very high kinesthetic modality. Group 3 has a profile that is characterized by an equal presence

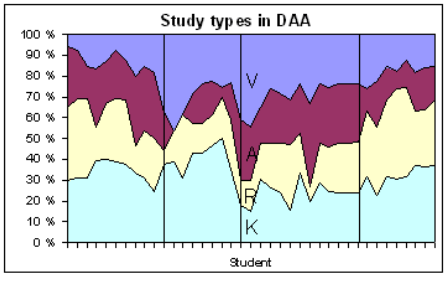


Figure 1: Normalized VARK profiles.

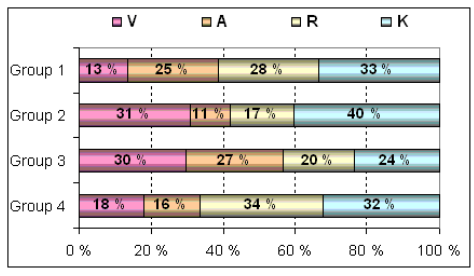


Figure 2: Average profiles of the clusters

Table 1: VARK profiles of academics.

	V	A	R	K
Teacher DAA, IMA	36%	29%	14%	21%
Assistant DAA,IMA	38%	6%	25%	31%
Teacher AI	18%	36%	18%	27%
Assistant AI	30%	17%	22%	30%

Table 2: Score percentages of the four groups.

	Finnish s.	DAA	AI	IMA
Group 1	**80**%	47%	54%	**90**%
Group 2	50%	50%	<48%	63%
Group 3	46%	70%	65%	**97**%
Group 4	50%	50%	**76**%	70%

Table 3: VARK and results correlations.

	DAA	AI	IMA
V	0.15	-0.18	0.05
A	0.12	0.14	0.47
R	0.01	0.47	-0.16
K	-0.33	-0.57	-0.44

Table 4: Correlations of the students' results.

	DAA	AI	IMA
DAA	1.00		
AI	0.48	1.00	
IMA	0.42	0.19	1.00

of all modalities with a preference toward the visual perception. In Group 4 profile, the R/W- and K-modalities are emphasized. All groups had an equal proportion of different nationalities; except for the Group 1, as seen in Table 2. The explanation can be that there were only few foreign students, distributed randomly across the clusters.

The performance of the students classified into the most separable four clusters is shown in Table 3. The scores are normalized on a scale from 0 to 100 percent; according to the present study regulations, a student with less than 48% of a maximal grade would not pass a course. The students in Group 1 and 3 have performed best in IMA course, and Group 3 students best in DAA course, and students of Group 4 performed best in AI course.

It can be observed that Aural types (the modality of Groups 1 and 3 have in common) performed well in IMA, Aural and Visual types (Group 3) performed best in DAA, and Read/Write students (Group 4) did well in AI course. A hypothetical student with the learning profile matching the Group 3 profile would perform well in all the courses, while the student taken from the Group 2, would likely experience problems with passing the courses in this study. It can be suggested that a student with a balanced profile, as seen e.g. in Group 3, survives best, since all the modalities are present and provide the student a capability to perceive and output the information in all four modes.

3.3 Comparison of VARK modalities with the course results

To complement the previous results, we analyzed the correlations between the VARK scores and the results of the whole population of students, as shown in Table 3. There was no significant correlation between DAA results and either V, A, R, or K scores, although the K variable was slightly negatively correlated. Interestingly, the correlations were found between the VARK scores and AI results: the R/W attribute was positively, and the K score negatively correlated. This suggests that the learners with higher preference in R/W got, in general, higher grades, while the learners of the K-modality got lower grades. In the IMA course, we observed a positive correlation between the Aural scores and the results and a negative correlation of the K-scores. All the mentioned correlations were of about the same magnitude.

It can be suggested that a good learner who performs well in some course, performs also well in another course. This is supported by the correlations between the final results of the courses computed for all the participants, see Table 4. The exception seems to be a relatively low correlation between the results of the IMA and AI courses showing that students who did well in the IMA course were more successful in DAA course than in the AI course.

Finally, we divided the students into four categories, V, A, R, and K, according to their strongest learning preference and compared the average performance of the four groups formed in this way, as shown in Figure 3. Not surprisingly, the students of same preference did

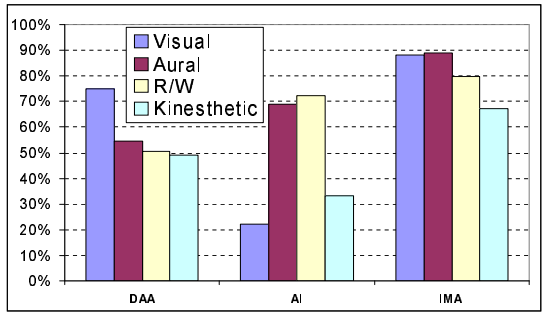


Figure 3: The mean performance of each learning type in the courses.

not perform equally in all courses. For instance, the learners whose strongest V-modality, performed best in DAA and second best (with an excellent grade) in the IMA course. When taking the AI course, the same students, in overall, would not pass. The learners with strongest modality was K would, although hardly, pass the DAA course and perform well in the IMA, but they would not pass the AI course. The students of Aural and Read/Write preference would perform on or above the average.

4 Conclusions

Our first hypothesis does not seem to be valid, as supported by the results. There is no best modality in general. The balanced VARK profile seems to indicate a better performance in all three courses in this study. Correlation analysis of VARK parameters and exam results verifies the previous observation: Aural students perform better in IMA course. Slight correlation between DAA and IMA results was observed: doing well in one indicates that a learner might perform well in another.

The claim of the second hypothesis seems to be supported by our results. The profiles of most successful students in a course match well with the profile of the teacher of the course. The teachers used different learning style and the courses were taught differently and these facts seemed to be reflected in the performance of same students attending the different courses. In our study, teachers had the strongest learning style preference in the Visual modality and students had the strongest preference in Kinesthetic modality. For both groups the proportions of A and K preferences were relatively similar.

References

Fleming, N. D., 2001. Teaching and Learning Styles: VARK Strategies. Honolulu Community College.

Fränti, P., 14.6.2004a. Design and analysis of algorithms course homepage. `http://www.cs.joensuu.fi/pages/franti/asa/`.

Fränti, P., 14.6.2004b. Image analysis course homepage. `http://www.cs.joensuu.fi/pages/franti/image/`.

Fränti, P., Kivijärvi, J., 2000. Randomized local search algorithm for the clustering problem. Pattern Analysis and Applications 3 (4), 358–369.

Gardner, H., Kornhaber, M. L., Wake, W. K., 1995. Intelligence: Multiple Perspectives. Harcourt College Publishers.

LaRussa, G., 14.6.2004. Artificial intelligence course homepage. `http://cs.joensuu.fi/pages/gaetano/artificial_intelligence.html/`.

Investigating Group and Individual Work Practices Using CASE Tool Activity Logging

Janet Hughes and Steve Parkes
University of Dundee

jhughes@computing.dundee.ac.uk, sparkes@computing.dundee.ac.uk

Abstract

This paper describes the application of CASE tool user activity logging to explore work practices of computing students during the software design process. The behaviour of students designing software in groups was compared to the behaviour of students designing software individually. The aim of the research was to identify teaching or learning approaches that may assist students to advance their abilities in software engineering. Results obtained confirmed observation studies that discernible differences exist between group working and individual working behaviour: groups explored design alternatives more than individuals whilst searching for a solution to a design problem, and made more use of dynamic modelling than individuals. It appears that group working practices have a parallel with advanced-level behaviour in software design, and should be encouraged in the curriculum.

1 Introduction

Evidence from literature shows that novices are less likely than experts to perform hypothesis testing and rejection, or to explore a design space for alternative solutions to a problem. Novice performers are more likely to be "satisficed" with a good-enough solution, and are less likely than expert performers to spend time evaluating work in progress. A study of computing students was performed to investigate if similar novice behaviour was exhibited during learning about the software design process. The motive for the study was to identify any teaching or learning approaches that could be employed to assist student object-oriented software developers to perform at a better level. An empirical approach was adopted, with multiple data sources being used in an exploratory case study. Initial participant observation studies and interviews indicated that both undergraduate and post-graduate students were insecure about their ability to perform the object modelling process well (Hughes, 2003). Furthermore, students discussed and debated design alternatives when in group working situations, but individuals working alone did not give major consideration to design alternatives, preferring to progress as rapidly as possible to the next stage of the design process without backtracking. The work described here represents a further part of the empirical research, utilising data logging to track the CASE tool actions of students as they performed analysis and design for software projects.

2 The Study

Observation research results may be influenced by a "Hawthorne effect", whereby participants' awareness of a researcher may prompt a reflexive response, resulting in events proceeding differently because of the observation. Automatic capture of data has been used occasionally as technique to bypass these methodological limitations whilst monitoring user actions, eg Ormerod and Ball (1993); Bowdidge and Griswold (1997). The intention of this study was to determine if the earlier contrasts observed between group and individual work practices could be corroborated using CASE tool user activity logging, whereby students' activities could be monitored whilst they were managing their own time, unobserved and not influenced by researchers or teaching staff.

2.1 Method and Participants

Keystroke and mouse click recording reduce the amount of time needed for observation, but produce large quantities of low-level event data which can be very time-demanding to analyse. Higher-level event data capture may help reduce the analysis effort. Researchers using this technique include Helms et al. (2000), whose raw data included date, time, and host system being used, and The Open University in the United Kingdom (Thomas and Paine, 2000) working to investigate how students learn to program in Smalltalk. Similarly, this study gathered high-level event data from a CASE tool used during the software design process. Participants were drawn from senior under-graduate students of Applied Computing at the University of Dundee involved in various projects over an eighteen-month period. Group work data capture took place within a ten-week two or three-person group project or an eight-week five or six-person group project. Individual work data capture took place four months later during a solo six-month final-year project.

2.2 Data Capture

Select Enterprise is a repository-based CASE tool that supports the use of the Unified Modelling Language (UML) for object-oriented analysis and design. Its object model contains a hierarchy of methods, properties, and events that control use of its interface, some of which are exposed so that a client application can use OLE automation to interrogate the properties of individual UML items in the repository. The user activity logging software designed by the authors ran on a user's computer alongside the CASE tool's client application, repeatedly interrogating the CASE tool client to determine the properties of the item currently selected by the user. The software tracked the life of the particular item being attended to by the user, such as an attribute or operation of a class. Text files with captured data were saved automatically at the conclusion of each session with author name, date, and time stamp. The sampling interval was one second.

3 Results

Evidence of how groups and individuals typically performed the design process was sought by analysing data logs to track the total time spent upon different modelling activities and their chronological sequence. In a study of how programmers use a program restructuring tool, Bowdidge and Griswold (1997) found evidence of groups saving an intermediate version of their programs to support the exploration of a design and subsequent backtracking. A related approach was used in this study: evidence of design exploration was sought by comparing components (projects, diagrams, classes, operations, and attributes) of participants' intermediate and final models. A reduction in the number of a component from intermediate models to the final model was taken as an indicator that alternatives had been considered before the final solution was concluded.

3.1 The Modelling Process

Proportionately, the greatest amount of group work time was spent modelling object interactions with sequence diagrams, although the full set of modelling activities was used. In contrast, individuals consistently spent the greatest proportion of time use case modelling and a very small amount of time was spent on state chart modelling (Table 1). Comparing the time proportions per activity using two-tailed t tests, only the difference between the means for object modelling was not significant ($t = 0.163$; $df = 5$; NS). The nature of the different types of project was investigated to explore if particular features of the projects contributed to these notable modelling proportion differences. Use Case Points (UCP) can be used to estimate project effort (Karner, 1993). Two-tailed t tests showed no significant

difference between the group and the individual projects in terms of the number of UCPs ($t = 0.1403$; $df = 5$; NS).

Table 1: Percentage of total time spent on different modelling activities.

	Use Case	Object Model	Sequence Model	State Transition
Group mean	6.60	18.75	52.85	18.33
Individual mean	47.43	17.47	27.93	0.60

The contrasting attentions given to different modelling activities was confirmed by a further analysis that compared the sequence of actions. Representative examples of charts showing sequences of modelling activities during a project are given in Figures 1 and 2.

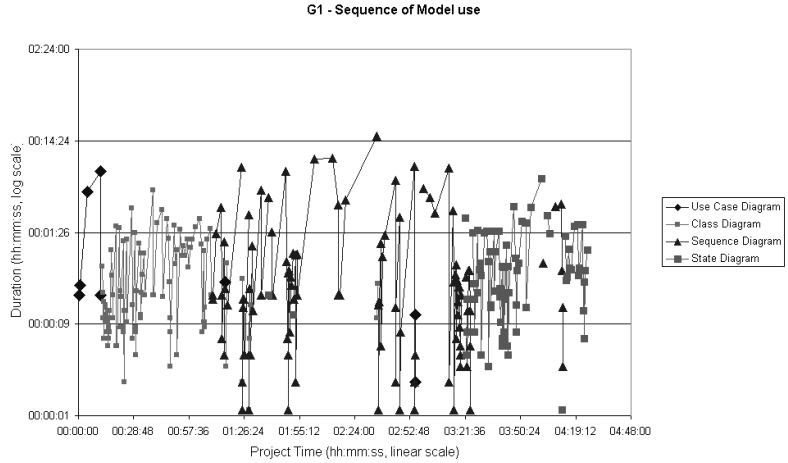


Figure 1: Example of sequence of modelling activities by group.

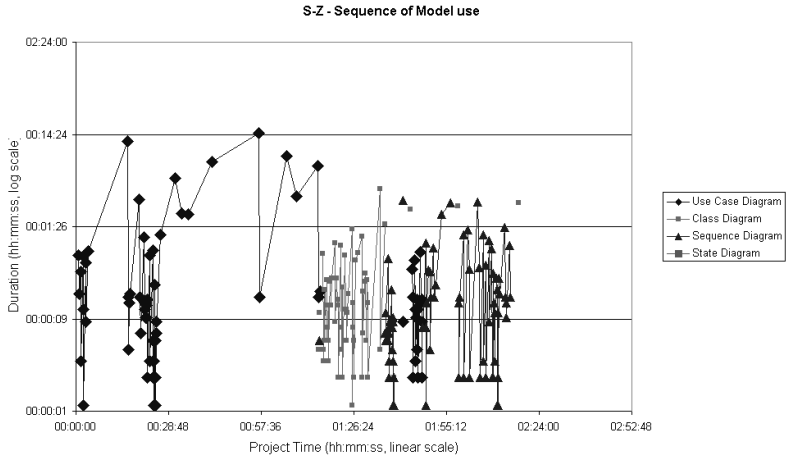


Figure 2: Example of sequence of modelling activities by individual.

It can be seen that dynamic modelling was a strong feature of the pattern of group working, particularly sequence modelling. Solo working was dominated by use case modelling, with a virtual absence of state chart modelling. The charts also revealed a minimal amount of backtracking to revise earlier models: a typical revision would have been that of object model amendment following testing of responsibility allocation during sequence modelling. Analysis

shows that the proportion of the sequence modelling time punctuated by object modelling using a class diagram by individuals was greater than that for groups, but that the extent of such intervention was very limited.

3.2 Design Exploration

Results showed that all participants had considered some alternatives, but that the extent of that exploration was small. More groups created alternatives than individuals, in terms of projects, class diagrams, sequence and state chart diagrams, classes, attributes, and operations. Only with use case diagrams did proportionately more individuals experiment than groups (Table 2). This data indicates that to a small extent alternative and competing designs were generated; considering component data in total, there is some indication that groups gave more attention to this than individuals.

Table 2: Greater Number of Alternatives Created.

	Project	Class Diagram	Sequence Diagram	State Diagram	Class	Attribute	Operation	Use Case
Group	*	*	*	*	*	*	*	
Individual								*

4 Discussion

4.1 Different Work Practices

Analysis of the actions performed with a CASE tool during software design has suggested some behaviour that is common to group working, and some common to solo working. Overall, the data corroborates observational and interview findings that groups and individuals performed the software development process differently, in the extent to which they explored alternative design models. Component data suggests that groups gave more attention to consideration of alternatives than individuals did. Results also showed that groups and individuals gave different attention to the various types of modelling activity, groups making greater use of dynamic modelling techniques than individuals did. The study supports the premise that group working practices have similarities to expert-like behaviour, in terms of exploration of alternatives and use of a full set of design modelling methods. Solo working does not appear to prompt such advanced-level behaviour. There is a growing amount of evidence in support of Extreme Programming as a methodology that produces high quality software. Possibly group design work has similarities with pair-programming, and teaching students to pair-design would encourage the application of advanced-level design skills, ie actively seek defects in suggested solutions, look for alternative solutions, and consider trade-offs between alternative solutions. Software engineering curricula naturally tend to focus upon the principles of object-oriented analysis and design, the details of a methodology such as that associated with UML, and practical skills involving the use of a CASE tool. Whilst mastery of the industry standard method of object-oriented analysis and design is clearly vital, it may be that curricula need also to emphasise more about design explorations and trade-offs, to help students to move away from a "satisficing" approach to software design.

4.2 Limitations of Data Logging Technique

A concern about the validity of the data logging results emanates from the voluntary nature of participation. A progressive decline in participation, from 93.8 to 63.6 percent, reduced the number of projects available for full investigation. Detailed analysis was made of seven projects

that showed complete logging: four group and three individual projects. Participant pressure did not threaten the validity of the study, but the sample size was small and self-selecting. Nevertheless, with regard to performance level, age range, and gender, the participants were found to provide a good representation of the full group. The scope of the data logging was satisfactory given that statistical generalisation of results to a wider sample was not an objective. Data was obtained in the natural circumstances of unsupervised laboratory work, from all utilised times of the day and week, and from projects tackled during a period in which laboratory arrangements remained unchanged.

5 Conclusion

Automated CASE tool data logging can provide a method of tracking group and individual actions during software design at any time of the day, thus providing data from otherwise unobserved situations and thereby removing the problem of errors due to sampling or the Hawthorne effect. In the study described, user activity logging provided evidence of the design process as performed by groups and by individuals. A number of contrasts in their CASE tool usage were noted, corroborating earlier findings from observational and interview studies. The results of the present study indicate that group working practices resemble advanced-level behaviour in respect of consideration of design alternatives. This suggests that tangible benefits may accrue from requiring team working to be a compulsory part of the software engineering curriculum. Curriculum designers may wish to consider (i) pair-design as a partner process to pair-programming, and (ii) if the design process should be assessed as well as design products. Further research is needed to repeat the study with a larger sample of students, and to investigate teaching methods that could be applied to focus attention upon design trade-offs, for example helping students to develop skills in evaluation and reflection, as well as technical knowledge.

References

Bowdidge, R. W., Griswold, W. G., 1997. How software engineering tools organise programmer behaviour during the task of data encapsulation. Empirical Software Engineering 2, 221–268.

Helms, J., Neale, D. C., Isenhour, P. L., Carroll, J. M., 2000. Data logging: Higher-level capturing and multi-level abstracting of user activities. In: Proceedings of the 44th annual meeting of the Human Factors and Ergonomics Society, San Diego, California, July 30 - August 4, 2000. Human Factors and Ergonomics Society.

Hughes, J. M., 2003. Modelling techniques to improve the learning & teaching of object-oriented design. Ph.D. thesis, University of Dundee.

Karner, G., 1993. Metrics for objectory. Cited in Schneider, G. and Winters, J.P. (2001). Applying Use Cases, Second Edition. Addison-Wesley, Boston.

Ormerod, T. C., Ball, L. J., 1993. Does programming knowledge or design strategy determine shifts of focus in prolog programming? In: Cook, C., Scholtz, J., Spohrer, J. (Eds.), Empirical Studies of Programmers: fifth Workshop. Ablex Publishing Corp., Norwood, N.J., pp. 162–186.

Thomas, P. G., Paine, C. B., 2000. Tools for observing study behaviour. In: A.F., B., E., B. (Eds.), 12th Workshop of the Psychology of Programming Interest Group, Cozenza, Italy.

ACE: Automated Compiler Exercises

Leena Salmela and Jorma Tarhio
Helsinki University of Technology

lsalmela@cs.hut.fi, tarhio@cs.hut.fi

Abstract

Automatic assessment has become popular in large introductory courses in computer science. Although the main issue has been to reduce the work load of course personnel also other benefits like better and faster feedback have been attained. We have constructed an automatic assessment system for home assignments to be used in a Compilers course. The system also features a graphical environment for doing the assignments.

1 Introduction

Home assignments have been a part of the Compilers course in Helsinki University of Technology for several years. The home assignments have covered finite state automata, parsers, generation of intermediate code, flow analysis and register allocation. The purpose of these assignments has been to activate the students during the term.

However, several problems have arisen with the assignments. First of all, given the current resources it has not been possible to give individual feedback to one hundred students and the feedback has arrived too late. Plagiarism has also been a growing concern in the course since all students have had the same assignments. Some students have also found the current practice of returning the home assignments as text files in a specified form unintuitive.

To attack these problems we are automating those home assignments that deal with finite state automata and parsers. Automatic assessment would allow for immediate feedback to the students and it would be possible to give individual assignments to students thus alleviating the problem of plagiarism. The system also contains a graphical environment for studying and completing the assignments. The visualizations are mostly adopted from JFLAP (Cavalcante et al., 2004) which is a visualization tool for formal languages and automata theory. JFLAP is based on earlier work of Susan Rodger.

Automatic assessment has been successfully used in introductory courses in the Helsinki University of Technology (Malmi et al., 2002). For example the Ceilidh system (Benford et al., 1993) and Scheme-Robo (Saikkonen et al., 2001) have been used in the programming courses, the TRAKLA2 system (Korhonen et al., 2003), which has a graphical interface for doing algorithm simulation exercises, has been used in the Data Structures and Algorithms course and the Stratum framework (Janhunen et al., 2004) has been used in several courses in the laboratory of Theorethical Computer Science. Automatic assessment has proved to be effective in these cases and the student response has also been generally positive.

Several visualizations of finite automata and parsers have been developed. Some of these visualization tools, like JFLAP (Cavalcante et al., 2004) and Exorciser (Tscherter et al., 2002), have taken a step towards automatic assessment. They allow the student to try building his own solution. Once the student thinks he has accomplished the task the tool will assess the solution and tell the student if it is right. These tools also allow the student to take a look at the model answer. However, these tools do not fully cover the assignments we have used and they are intended for self study so they do not keep track of students' points and solutions.

The rest of this paper is organized as follows. In Section 2 we give an overview of the assignments to be automated. In Section 3 we describe the ACE system and in section 4 we give some concluding remarks.

2 Overview of the Assignments

The assignments of the course have been organized into six rounds. The first three rounds handle the front end of the compiler and the last three rounds the back end. We are now attempting to automate only the first three rounds, partly because the course will be split into two courses in the future, and the former part will get an even larger attendance while latter part will attract less students. The first three rounds deal with finite state automata (FSAs), LL parsing and LR parsing respectively. Each round has four assignments.

In the first assignment of the FSA round the student is given a regular expression and his task is to form a nondeterministic finite state automaton (NFA) using Thompson's construction. Then in the second assignment the constructed NFA is simulated with a given input. In the third assignment the NFA is converted to a deterministic finite state automaton (DFA) and this DFA is then simulated in the last assignment of this round.

The second round deals with LL parsing. First the student should remove left recursion from the given grammar. In the second assignment this grammar is left factored. Then in the third assignment the First and Follow sets needed in the LL parse table construction are calculated. Also the filling of the parse table is part of the third assignment. Then in the last assignment the constructed parser is simulated with the given input.

In the third round an LR parser is constructed. In the first assignment the student forms the LR(0) item sets of a given grammar and figures out the transitions between them. Then in the second assignment the First and Follow sets are calculated. Based on these sets also the LR parse table is constructed. The grammar used in this exercise is ambiguous so the parse table now contains ambiguity. In the third assignment of this round the ambiguity is removed from the parse table so that the given precedence and associativity constraints are satisfied. In the last assignment the constructed parser is simulated with the given input.

Some of these exercises are clearly algorithm simulation exercises. The simulation of a FSA or a parser clearly falls into this category. The solution to this kind of exercise is an ordered list of steps. Some of the other exercises include simulation of an algorithm but the algorithm is more loosely defined. For example the Thompson's construction algorithm does not define a total order for the construction of the automaton parts. Thus it only defines a partial order for the steps that are needed to construct the whole automaton. Of course a total order may be enforced in such an algorithm but this would unnecessarily complicate the assignment. Some of the assignments are even more loosely defined like the removal of left recursion from a grammar. In this case some transformation rules are presented in the study material but the use of exactly these rules is not enforced. These exercises are conceptual in nature. They test the student's understanding of the concept rather than knowledge of a specific algorithm.

We have around ten assignment sets for each round. A set for each student is chosen randomly among those sets. Moreover we allow permuting and replacing of local strings and names in the assignments in order to artificially increase the number of different assignments. We are also studying ways to generate new grammars and regular expressions for assignments.

3 ACE

Given the various types of exercises that the automatic assessment system needs to support we decided to build a client for doing the exercises and verifiers for checking them. These components could then be embedded into a framework which takes care of submissions and the needed book keeping. We call the client and the verifiers Automated Compiler Exercises (ACE).

Here we give an overview of the client and the verifiers. We have embedded these in a framework called Stratum (Janhunen et al., 2004) which has been developed in the Laboratory of Theoretical Computer Science in Helsinki University of Technology.

3.1 ACE Client

The central part of the ACE client is the visualization of the data structures needed in the assignments. The client needs to support the following visualizations:

- Editing and simulating a FSA. When converting a NFA to a DFA or building the LR item set automaton special restrictions apply to editing the labels of states.
- Editing a grammar.
- Defining the First and Follow sets for a grammar.
- Editing LL and LR parse tables.
- Showing precedence and associativity information of operators.
- Simulating LL and LR parsers.

Almost all these features are present in the JFLAP software (Cavalcante et al., 2004). Thus the ACE client is built reusing the code from JFLAP. Some changes of course needed to be done. JFLAP does not support showing precedence and associativity information for operators so visualization for this was built. The simulation of FSAs and parsers in JFLAP are merely animations so we needed to add some interactivity here so that the students can show how the FSA or parser works. For example when simulating an LL parser using ACE the student has two choices in each step. He can either choose to advance in the input or apply a rule from the parse table.

Another major change was adding the notion of assignment rounds and assignments. Now ACE can lead the student through a assignment round one assignment at a time. Other changes included the design of a new file format which contains the information about assignments and assignment rounds. Because of the new file format it is also not so easy for the students to use JFLAP to generate the correct answers. The generation of correct answers was of course disabled from the user interface. In the end the possibility to submit an assignment was added to ACE when it was embedded to the Stratum framework.

Fig. 1 shows a screenshot of the ACE client. Here the user is converting a NFA to a DFA. He has already defined the initial state of the DFA and the state which the DFA enters after reading the symbol 'x' in the initial state. The labels of the DFA states show their corresponding NFA state sets.

3.2 Verifiers

We were also able to reuse some parts of JFLAP when building the verifiers. Some of the assignments like removing left recursion from a grammar are not supported by JFLAP so we needed to implement whole verifiers for these. The simulation of FSAs and parsers in JFLAP are only animations without the possibility of error so we needed to implement new verifiers for these too.

The verifiers have the following general structure. First they check if the input the student used was the one given in his assignment. Then they generate the model answer and compare that to the student's solution. As a last step the verifiers generate feedback to the student.

It would be nice to check some of the exercises in a bit different way. For example we would like to check the removal of left recursion as follows. First we would check that there is no left recursion left in the grammar produced by the student. Then we would need to check that the grammar still produces the same language. This approach is unfortunately not possible because it is undecidable to determine if two context free grammars produce the same language. So in this case we have to enforce the use of a set of transformation rules to be able to check the exercise.

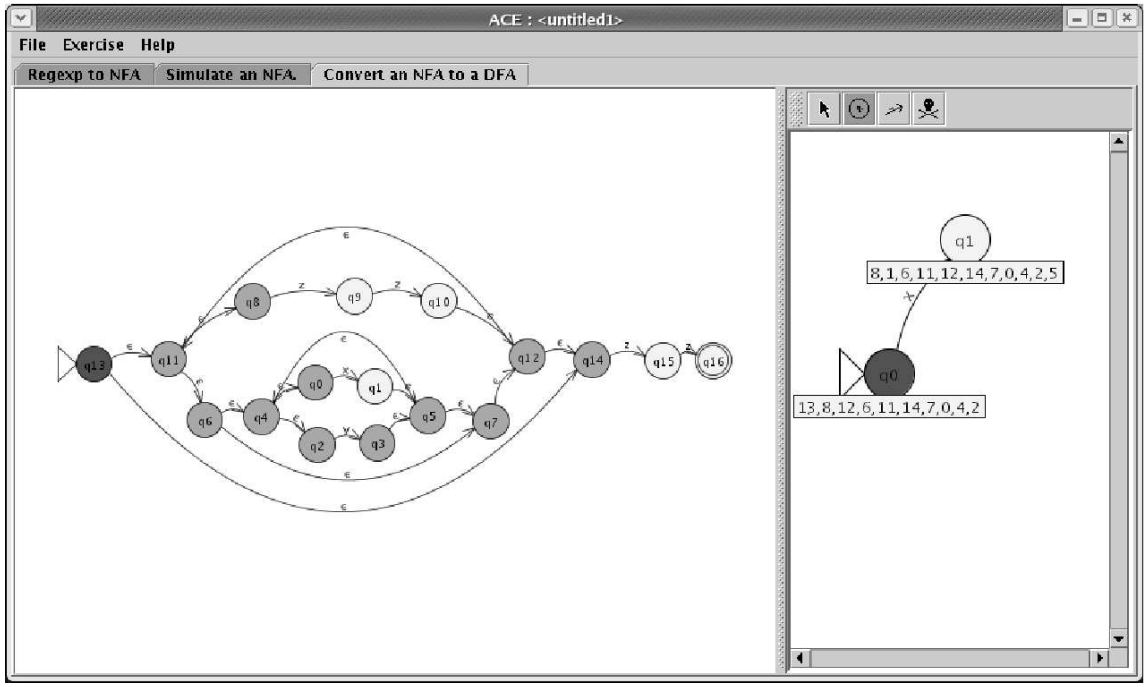


Figure 1: A screenshot from the ACE client. Here the user is converting a NFA to a DFA.

4 Concluding Remarks

We have described the ACE system for automatically assessing assignments related to finite state automata and parsers. The system supports individual assignments for students and it has a visual interface for doing the assignments.

The system has been used this fall in our Compilers course. We will report detailed experiences and results later on. The first impression is that our students found the system appealing. When considering the activity of students in our course in 2002–04, this fall we noticed an increase in the number of students working on the third assignment which is a voluntary exercise for most students.

5 Acknowledgements

We thank Susan Rodger and Tomi Janhunen for letting us use their codes.

References

Benford, S., Burke, E., Foxley, E., Gutteridge, N., Zin, A. M., 1993. Ceilidh: A course administration and marking system. In: Proceedings of the 1st International Conference of Computer Based Learning.

Cavalcante, R., Finley, T., Rodger, S. H., 2004. A visual and interactive automata theory course with JFLAP 4.0. In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education. ACM, pp. 140–144.

Janhunen, T., Jussila, T., Järvisalo, M., Oikarinen, E., 2004. Teaching Smullyan's analytic tableaux in a scalable learning environment. In: Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education.

Korhonen, A., Malmi, L., Silvasti, P., 2003. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In: Proceedings of the 3rd Finnish/Baltic Sea Conference on Computer Science Education. pp. 48–56.

Malmi, L., Korhonen, A., Saikkonen, R., 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In: Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education. ACM, pp. 55–59.

Saikkonen, R., Malmi, L., Korhonen, A., 2001. Fully automatic assessment of programming exercises. In: Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education. ACM, pp. 133–136.

Tscherter, V., Lamprecht, R., Nievergelt, J., 2002. Exorciser: Automatic generation and interactive grading of exercises in the theory of computation. In: 4th International Conference on New Educational Environments. pp. 47–50.

Some Ideas on Visualizations of Abstract Execution to Provide Feedback from Programming Exercises

Petri Ihantola
Helsinki University of Technology
Department of Computer Science and Engineering
Finland
ptenhune@cs.hut.fi

Abstract

Visualizing abstract execution, execution of programs without precise knowledge about the input, is a rarely studied subject in the field of software visualization. In this paper some new ideas on how to use visualizations of abstract execution to provide feedback from programming exercises are presented.

1 Introduction

As typical software has grown more and more complex, systems that enhance, simplify and clarify the mental representation of the software have become more and more important. Such systems are often called *software visualization* (SV) systems. These systems are nowadays commonly used in computer science education as well as in professional software engineering.

Visualizations are always based on some sort of analysis. Analyzers, on the other hand, seem to be focusing towards extremes, where absolutely nothing about the input is known or the input is exactly known. However, algorithms and programs are often used in a domain where additional information about the typical input is available (e.g. string search from an English text, rather than from random data). We believe that there is need for SV systems, where the input is abstractly expressed, but not completely open. Static analysis and especially abstract execution can be used to obtain this goal. From analyzers and SV systems there is a small leap into automatic assessment and feedback.

Abstract execution (aka *abstract interpretation*) is a group of static analysis techniques, predicting runtime behavior and properties of programs. The techniques are based on abstractly expressed input; like evaluating $f(x, y) = x - y$ with values X and $2X$ rather than with real values like 1 and 2, in mathematics. Abstract execution has strong mathematical foundations (see e.g. Nielson et al., 1999) and an essential role in numerous industrial applications. For example many *compiling errors*, excluding syntax errors, are found with abstract execution. Formal verification (i.e. model checking) of programs, highly standing on abstract interpretation, has also been used to detect *runtime errors* (see e.g. Havelund and Skakkebæk, 1999). Unfortunately many real-life applications are too complicated for this approach.

Even though the importance of abstract execution is evident, Diehl (2001) has pointed out that SV with abstract execution has been a nearly unresearched topic. This observation originally raised the author's enthusiasm towards the topic on hand. Diehl's observation seems to be still valid: In the latest Program Visualization Workshop (Korhonen, 2004) there were 20 papers and only one[1] was concentrating on results obtained through abstract execution (i.e. data-flow analysis). In this paper we present ideas from the author's ongoing Masters thesis project: how abstract execution and SV could be used to provide different feedback from programming exercises. Our focus is on imperative languages.

The rest of this paper is organized as follows: Section 2 is about the role of abstract execution in the field of SV. A couple of existing visualization systems have been selected to illustrate the results of previous work. Some common design principles of these kind of systems are also described. Section 3 discusses about some limitations of current SV systems

[1] Papers were classified by reading the abstract and if necessary also introduction and conclusions. Because final papers were not yet published copies distributed during the workshop were used.

based on abstract execution and presents new ideas on how abstract execution could be used to provide feedback from programming exercises. Finally section 4 shortly summarizes this all.

2 Visualizing Abstract Execution

In SV systems, by *visualizing abstract execution*, we mean systems clarifying mental models about program dynamics in where the analysis is based on abstract domains. Our motivation to do this raises from the observation of Braune and Wilhelm (2000): "Badly chosen input data does not explain the essentials of the algorithm." The same can be extended to programs in general and should also be kept in mind when automatically assessing programming exercises.

In the following, we will first describe how abstract execution of computer programs is placed in the field of SV. After that two existing SV systems and design principles are presented to give an overview about the previous work.

2.1 Diehl's Classification

Diehl (2001) organizes what can be visualized in the field of SV (according to the selected abstraction level and view point) into the following four times four matrix.

$$\{\text{Software Systems, Algorithms/Programs, Abstract Machines, Hardware Native Code}\} \times \{\text{Static Structure, Concrete Execution, Abstract Execution, Evolution of Static Structure}\}$$

We are interested in $\langle$ Algorithms/Programs, Abstract Execution$\rangle$. However, Diehl classifies data flow and control flow graphs to present only static structure (i.e. there is no dynamics present). Our point of view is that because a path in a control flow graph can present a possible trace from real execution, flow graph could also be used to visualize dynamics. Moreover, if the input is abstractly expressed, some branches from flow graphs can be pruned by the means of abstract execution. This is our justification for the fact that abstract execution is mentioned in the title, but flow and dependency graphs are taken into discussion.

2.2 Visualization Systems

Alexsa (Bieber, 2001) is designed to support algorithm explanation through shape graphs (Wilhelm et al., 2000). The system is built on the TVLA system (Lev-Ami and Sagiv, 2000). The idea (Braune and Wilhelm, 2000; Wilhelm et al., 2001) itself was presented before Alexsa was implemented. Each execution point in the source code is automatically annotated with an abstract shape graph. This graph illustrates all the possible memory (heap) configurations, in that point of execution. This approach has a benefit that from dynamic memory structures the parts that are modified or accessed are emphasized.

CodeSurfer (Anderson et al., 2003a,b) is a commercial tool for software developers. It uses program dependency graphs (Ferrante et al., 1987) and pointer analysis to provide code visualizations with versatile highlighting and interactive navigation within the code. There is also a possibility for model checking so commercialism is balanced with versatile features. The tool is an example from novel textual visualizations.

There are several other systems mostly concentrated on graphically visualizing data and control flows. Alexsa and CodeSurfer were selected because they both have academic background and together these provide different viewpoints (i.e. graphical *vs.* textual) into SV and analysis techniques (i.e. shape analysis *vs.* flow/dependency graphs).

Certain design principles seem to be common for all SV systems in this field. Modularity has an essential role, as pointed out by Anderson et al. (2003a). There should be a language dependent *front-end*, one for every supported language. Front-ends are used to provide input

for the *analyzer*. Analyzer itself should be language independent. Finally there should be *visualization modules* to view the results. Most abstract execution techniques are complex, leading analyzers to be highly specialized. Therefore, several analyzers for different uses are needed. *Analyzer generators* are used to simplify the process of creating these complex modules. For example AbsInt[2] offers a full product family with described modularization.

3 Discussion

3.1 Unknown Input *vs.* Abstract Input

As pointed out in the introduction the focus of SV seems to be on completely unknown or exactly known input. We believe that following, commonly known, approaches can be used to move towards analyzers where the input is abstractly expressed, but not completely open:

- Symbolic execution (King, 1976) uses *symbolic values* as input. If the analyzed method takes several arguments, these can be made dependent on each other (i.e. same variable can occur in several symbolic expression standing for different arguments).

  Symbolic execution is based mainly on constraint propagation, so it is at its best within integer domains. However, recently the analysis has been extended to deal with dynamically allocated structures (Khursid et al., 2003) and arrays (Visser et al., 2004).

- Constraints are often more powerful than symbolic expressions themselves. For example the fact that the length of an array A is no longer than 6 could be expressed with unary constraints $\{A.length < 7,\ A.length \geq 0\}$.

- Annotation (e.g. especially formatted comments) is a commonly used technique in abstract execution. It can be used to add information directly into the source code (e.g. constraints for the return values of methods and upper limit for how often a certain loop is performed). This approach might be extremely tedious, but it considerably reduces the complexity of the results of abstract execution and therefore makes results easier to understand.

3.2 Providing Feedback from Programming Exercises

Static analysis has also been used to provide feedback from programming exercises (see e.g. Truong et al., 2004). The feedback has been based on style or complexity (e.g. cyclomatic measures compared to the model answer). In the following we describe two (programming) exercise types where the feedback is based on abstract execution.

- Let us consider small programming exercises where students are asked to implement a simple procedure returning a value of a verbally or mathematically defined function. This kind of exercises can be used to introduce control flow related statements for students.

  In general, equivalence of programs is undecidable. In extremely simplified cases, however, results obtained through symbolic execution might be used to compare programs or methods. Thus, this could be used in the assessment of small programming exercises. If the student's solution strategy is remarkably complex or elegant it can be problematic to compare symbolic model answer and symbolic student's solution. We are planning to address this problem by comparing cyclomatic complexities between student's solution and model answer in order to identify such cases.

  Constraints and symbolic values mentioned in the previous subsection can also be used to create predefined (abstract) test cases. With *abstract test cases* symbolic execution

[2] http://www.absint.com/

can be used to provide feedback such as: "if your method takes arguments $X > 0$ and $Y > 0$ the result is X^{Y-1} when it should have been X^Y".

- Automated test input generation is commonly used in unit testing to find a set of tests so that e.g. a branch coverage is reached. A predefined set of tests can be used as a basis in which the tests are added. We are planning to use this strategy to compare the student's answer against the specification or the model answer like in e.g. SchemeRobo (Saikkonen et al., 2001).

  The novelty comes in feedback. Tests leading into unexpected behavior are not directly revealed as a feedback. Data and control flow graphs related to failed tests are graphically visualized, with corresponding paths highlighted. Shape graph visualizations à la Alexsa can also be combined into these graphs. We believe that this approach (at least in some cases) gives sufficient information about where the error is, but also forces students towards active learning.

Visser et al. (2004) have used symbolic execution to create tests with high code coverage. During the symbolic execution, constraint satisfaction problems (CSPs) are built. Every conditional branch splits the CSP into two and adds a new constraint into both CSPs. If the exercise is a *fill in exercise*[3], annotations in the skeleton code can also be used to generate constraints. Finally CSPs are solved to obtain the real test inputs[4]. Aforementioned test input generation and testing can be used in automatic asessessment of programming exercises. The feedback can be a combination of flow graphs with path highlighting and a CSP from where the failed test was derived. This CSP can be understood as an abstract test case, which failed.

These approaches do raise the abstraction of the feedback. This can be a good thing, because it forces students to think and therefore activates learning. Downside is that the feedback can be too abstract for some learners. We believe that a combination of exact feedback about failed tests and more abstract feedback should be used together. The effective impact of different types of feedback should be further researched.

Important fact to remember is, that there is no silver bullet for software testing or automatic assessment of programming exercises. Rice's (1953) theorem shows that interesting properties (e.g. halting property) about the program are undecidable. However, we believe that in some simple cases, that is what programming exercises typically are, combination of these two can be successfully used to provide educationally effective and reasonably accurate feedback for students.

4 Summary

Techniques from software testing are proposed to be used in the automatic assessment of programming exercises. A combination of concrete and abstract execution has seen promising to provide different feedback (software visualizations) from programming exercises. However, more research is needed.

5 Acknowledgments

I'm doing my Thesis in the guidance of Ari Korhonen and under the supervision of Lauri Malmi. Otto Seppälä and Jussi Nikander gave many valuable comments about this paper.

References

Anderson, P., Reps, T., Teitelbaum, T., August 2003a. Design and implementation of a fine grained software inspection tool. IEEE Transactions on Sowtware Engineering 29 (8), 721–733.

[3]For example a working program or class with only one method missing

[4]Only one solution for each CSP is needed.

Anderson, P., Reps, T., Teitelbaum, T., Zarins, M., Inc., G., July/August 2003b. Tool support for fine-grained software inspection. IEEE Software 20 (4), 42–50.

Bieber, R., 2001. Alexsa – algorithm explanation by shape analysis – extensions to the TVLA system. Diplomarbeit, Universität des Saarlandes, Saarbrücken, Germany.

Braune, B., Wilhelm, R., 2000. Focusing in algorithm explanation. IEEE Transactions on Visualization and Computer Graphics 6 (1), 1–7.

Diehl, S., 2001. Future perspectives – introduction. In: Diehl, S. (Ed.), Software Visualizations. Vol. 2269 of Lecture Notes in Computer Science (LNCS). Springer, Ch. 5, pp. 347–353.

Ferrante, J., Ottenstein, K. J., Warren, J. D., June 1987. The program dependency graph and its uses in optimization. ACM Transactions on Programming Languages and Systems 9 (3), 319–349.

Havelund, K., Skakkebæk, J. U., September 1999. Applying model checking in java verification. In: Dams, D., Gerth, R., Leue, S., Massink, M. (Eds.), Proceedings of 5th and 6th International SPIN Workshops. Vol. 1680 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, Toulouse, France, pp. 216–231.

Khursid, S., Păsăreanu, C. S., Wisser, W., April 2003. Generalized symboliz execution for model checking and testing. In: Garavel, H., Hatcliff, J. (Eds.), Proceedings 9th International Conference on Tools and Algorithms for Construction and Analysis. Vol. 2619 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 553–568.

King, J. C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.

Korhonen, A. (Ed.), 2004. Proceedings of the 3rd Program Visualiztion Workshop. To appear.

Lev-Ami, T., Sagiv, S., 2000. Tvla: A system for implementing static analyses. In: Proceedings of the 7th International Symposium on Static Analysis. Vol. 1824 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, pp. 280–301.

Nielson, F., Nielson, H. R., Hankin, C., 1999. Principles of Program Analysis. Springer Verlag.

Rice, H., 1953. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 83.

Saikkonen, R., Malmi, L., Korhonen, A., 2001. Fully automatic assessment of programming exercises. In: Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'01). ACM Press, Canterbury, United Kingdom, pp. 133–136.

Truong, N., Roe, P., Bancroft, P., 2004. Static analysis of students' java programs. In: Proceedings of the sixth conference on Australian computing education. Australian Computer Society, Inc., pp. 317–325.

Visser, W., Păsăreanu, C. S., Khurshid, S., 2004. Test input generation with java pathfinder. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. ACM Press, pp. 97–107.

Wilhelm, R., Müldner, T., Seidel, R., 2001. Algorithm explanation: Visualizing abstract states and invariants. In: Diehl, S. (Ed.), Software Visualizations. Vol. 2269 of Lecture Notes in Computer Science (LNCS). Springer, Ch. 5, pp. 381–394.

Wilhelm, R., Sagiv, S., Reps, T. W., 2000. Shape analysis. In: Computational Complexity. pp. 1–17.

A Visual Approach for Concretizing Sorting Algorithms

Ilkka Jormanainen
Department of Computer Science
University of Joensuu

ilkka.jormanainen@cs.joensuu.fi

Abstract

Algorithm visualization is an efficient way to teach programming. The Concretization Environment Framework (CEF) combines algorithm visualization with concrete objects (e.g., Lego Mindstorms robots). CELM, Concretization Environment for Lego Mindstorms, is an application of this framework. By using the framework, the user can turn the mental model the user has, into a concrete one. User feedback on the framework and its application has confirmed the functionality of the concept.

1 Introduction

One of the main difficulties that students of computer science face is understanding algorithms. Traditionally, algorithms have been taught verbally and with the use of blackboard or slides. With these, it is only possible to visualize algorithms in a static way. In past decades, researchers have developed different kinds of systems for algorithm visualization. Most of these systems allow the user to interact with the visualization and the algorithms are often visualized through animation (Stasko (1990); Ben-Ari et al. (2002)).

Robot technology has become cheaper and has been adopted widely to teach programming and computer science especially to novices. Robotics has been used to motivate students to learn programming. A student can create concrete new knowledge and learn in constructionist way by interacting with real world objects (Ben-Ari, 1998). This interaction can also lead more to hands-on learning with algorithms. In *algorithm concretization*, the algorithm's execution is emulated by robotics or other real world objects. In this way, robots engage the student with the algorithm and, thereby, foster learning.

This paper is based on my Master's Thesis (Jormanainen, 2004). In this paper, I will present the framework for concretizing algorithms and present an implementation of the application that is based on the framework. With this application, the user is able to make concretizations for sorting algorithms more easily than it has been done in (Gonzalez, 2004), which is basis of this work. Furthermore, I will present a novel idea about *role-based concretization*, which can be used with the application.

2 Background: Concretizing Bubble Sort algorithm with Lego Mindstorms

In his research, Javier Gonzales developed some concretizations of sorting algorithms with Lego Mindstorms (Gonzalez, 2004). The main idea in his work was to use a *master* robot that controls other robots (*slaves*). In this scheme, every robot has an individual id and a weight. These pieces of information are used to sort the robots with a sorting algorithm. Algorithms for the robots were developed in NQC (Not Quite C), which is a C-like programming language for Lego Mindstorms robots (Baum et al., 2000), and in Java with LeJOS. For more information about implementation, see Gonzalez (2004).

As it can be seen in Gonzalez (2004), the implementation is complicated. This makes it difficult to use concretization in an efficient way, for example, when teaching programming or algorithms to novices. It needs a long code to implement even a simple sorting algorithm, like Bubble Sort (see Gonzalez (2004)). However, as it has been stated in Gonzalez et al. (2004), this method of teaching is promising and worth further study.

3 The Aim of the Project

The main goal of this project was to develop an application that may help the user (teacher, instructor) construct concretizations for sorting algorithms. The user defines the interesting events of the algorithm and concretizations for these events. The user can use, for example, *role-based concretization*.

Role-based concretization is a novel way to define concretizations with robots. The concept is based on idea that the data of a program or an algorithm has a certain *role*. It has been found that the following list of roles of variables covers 99% of all variables in novice-level programs: Constants, stepper, follower, most-recent holder, most-wanted holder, gatherer, one-way flag, temporary and organizer (Sajaniemi, 2002). With these roles, it is possible to define a representation for each variable in the program.

However, in this application we see the concept of role in a different way. Gonzalez (2004) has defined two roles for concretizing sorting algorithms: left and right. In this case, the role is defined based on the physical position of the robot or other object. In Figure 1, `Item 2` has the role *left* and `Item 5` has the role *right*. During the execution of the algorithm, the robots make pre-defined behaviours based on these roles.

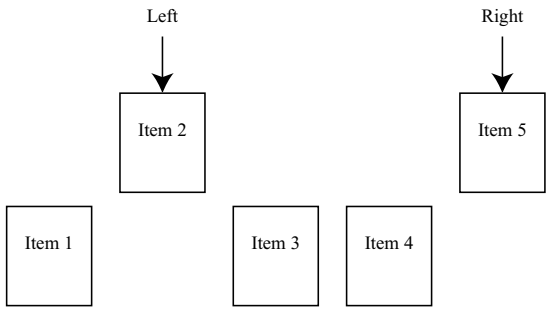


Figure 1: Roles left and right are defined by their physical position.

Concretizations are defined by dragging robots onto a specified area in the application. After that, the user allows the application to upload the code to the robots. Then, the robots execute the concretization. In this way, the mental models that users have in theirs minds, become concretized in the material world (Figure 2).

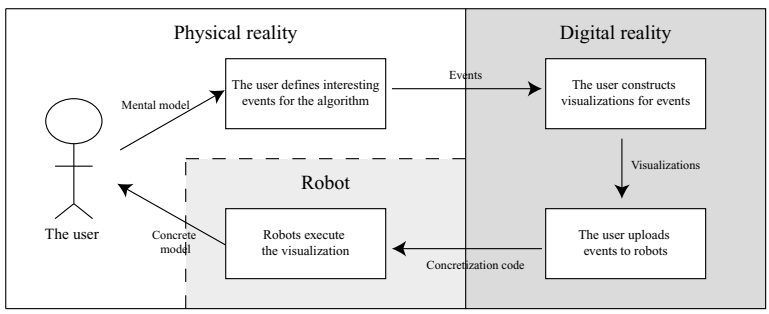


Figure 2: The user's mental models become concretized in the material world.

4 Design and Implementation of the Framework and the Application

The architecture of the framework contains three separate layers. For each layer, there is some output which serves as input in the next layer. Communication between layers is bi-directional. This means that physical objects can communicate and send information about their states to the application. In this way, it is possible to track the movement of the robot. Figure 3 presents this structure and communication between layers.

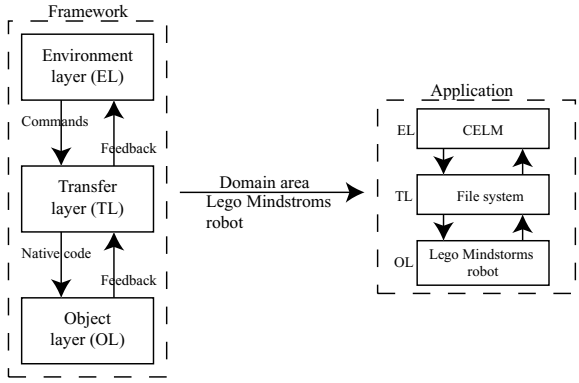


Figure 3: The relation between the framework and the application.

At the moment, the whole system is implemented in Java. However, it is also possible to produce any layer of the framework with some other programming language. The most important issue is to ensure that the layers give output in the right format. Also, each layer has to have the capability to use the output of the previous level as its input. These layers can be replaced with another one. When replacing the object layer, it might be necessary also to replace the transfer layer, or part of it. However, one can replace the whole transfer layer or part of it without changing the object layer or the environment layer at all.

To achieve full support for these features, definitions for interfaces between layers must be developed further. Especially, interfaces from the object layer to the environment layer (via the transfer layer) have to be developed carefully in the future. For this project, I decided on an interface from the environment layer to the transfer layer. This interface contains codes that the application at the environment layer has to produce. The transfer layer has to have the capability to transfer this code to the native code for robots (or other objects) at the object layer.

The environment layer contains an application that is dedicated to the control of one or more robots. With the application, the user can construct movements and other behaviours for robots, and send them to the robot (Figure 4). The application has been implemented purely with Java, so it can be used in diverse platforms, such as Windows, Linux or Macintosh. The only requirement is the need for a LeJOS environment, which is used in the transfer layer to compile the code produced by the environment layer to the native code of the object layer.

5 Conclusion and future work

In this paper I have presented a concept for algorithm concretization, which is based on research done at the Department of Computer Science, University of Joensuu (Gonzalez et al., 2004). However, there is a need for a framework and for an application which a user (for example a teacher or other instructor) can use to produce concretizations. To fulfill this need, I developed a framework that, makes it possible to use diverse platforms in an easy way. In this paper, I have presented the framework and one possible application for it. Furthermore, I have described a novel concept, *role-based concretization*, which can be used when designing concretizations for algorithms.

To assess the potential usefulness of this approach, I conducted a questionnaire in which I asked for opinions and suggestions about the application and the framework. Six researchers in computer science education answered the questionnaire (Jormanainen, 2004). According to the answers, the application and robotics can be used to illustrate abstract concepts, such as sorting and searching algorithms, especially when teaching children or novices. Answers also indicated that the concept *role* can be used with this approach. However, some doubts about the approach were presented.

There were some very concrete suggestions for improving the framework and application.

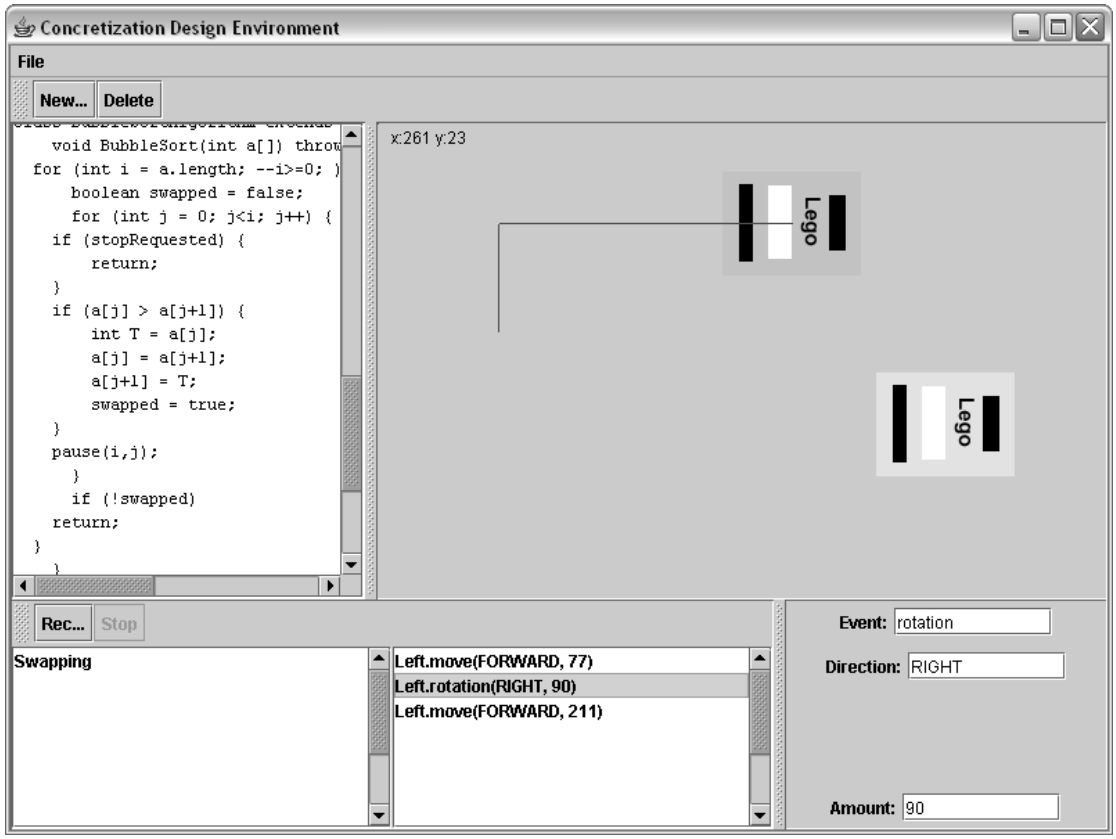


Figure 4: The user interface of the application.

A interesting approach could be to combine concretization with robotics and Empirical Modelling (EM). The Empirical Modeling is an approach for constructing computer based models, that can assist in the understanding of a phenomenon (Roe, 2003). For example, a simulator that could observe the behaviours of robots, based on EM, could be a very interesting topic for further study.

References

Baum, D., Baum, D., Gasperi, M., Hempel, R., Villa, L., 2000. Extreme Mindstorms: an Advanced Guide to LEGO MINDSTORMS. APress.

Ben-Ari, M., 1998. Constructivism in computer science education. SIGCSE Bulletin 30 (1), 257–261.

Ben-Ari, M., Myller, N., Sutinen, E., Tarhio, J., 2002. Perspectives on program animation with Jeliot. In: Diehl, S. (Ed.), Software Visualization. Vol. LNCS 2269 of Lecture Notes in Computer Science. Springer-Verlag, pp. 31–45.

Gonzalez, J. L., 2004. Software Visualization with Lego Mindstorms. Master's thesis, Department of Computer Science, University of Joensuu, Finland.

Gonzalez, J. L., Myller, N., Sutinen, E., 2004. Sorting out sorting through concretization with robotics. In: Proceedings of the working conference on Advanced visual interfaces. ACM Press, pp. 377–380.

Jormanainen, I., 2004. A Visual Interface for Concretizing Sorting Algorithms. Master's thesis, Department of Computer Science, University of Joensuu, Finland.

Roe, C., 2003. Computers for Learning: An Empirical Modelling perspective. Ph.D. thesis, Department of Computer Science, University of Warwick, United Kingdom.

Sajaniemi, J., 2002. An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In: Proceedings of IEEE 2002 Symposia on Human Centric Computing Lanuguages and Environments (HCC'02). IEEE Computer Society, pp. 37–39.

Stasko, J., September 1990. TANGO: A Framework and System for Algorithm Animation. IEEE Computer 23 (9), 27–39.

Program Code Emphasizing Tool for Learning Basic Programming Concepts

Sami Mäkelä and Ville Leppänen
University of Turku and TUCS, Department of Information Technology

sajuma@utu.fi, ville.leppanen@it.utu.fi

Abstract

We present a tool that can be used to emphasize parts of Java programs having desired syntactic or semantic meaning. The tool identifies a rich set of program entities, e.g. formal argument definitions, constructors, and calls of static methods. The tool helps students to learn basic programming concepts as well as language constructions.

1 Introduction

Learning to program is difficult. Learning the meaning of basic concepts and constructions is essential basis for the ability to write new programs. Based on practical teaching experience, we claim that many students fail to understand how the constructions appear and concepts are applied in large programs. The topic of this paper is a tool that aids in the learning process by highlighting requested constructions and concepts from arbitrary Java programs.

In the early phases of teaching how to program, students are usually taught of which components programs are made, what concepts are related to the programming language constructions, and what kind of syntax of the constructions have. Learning partially happens by studying programs or partial programs made by others and trying to understanding the meaning of programming (language) concepts with respect to those examples. After the first phase, teaching often proceeds to making own programs, learning to solve small problems, mastering complicated concepts of OO languages (inheritance, generics, ...), and learning to use the library.

Our tool, *JavaEmphasizer* (see Figure 1), aids in the first part of the learning process by emphasizing requested parts of chosen programs. The tool can emphasize purely syntactic constructions (e.g. formal arguments of methods, default-branch of switch statement, constructor) and well as concepts having semantical meaning (e.g. places where instance variable is used, call of a static method). The emphasizer has HTML and LaTeX backends.

We believe that the tool can greatly aid in the learning process, because there are quite a lot of concepts and constructions to learn in modern object-oriented programming languages. Learning by example is an effective method, but the large size of useful programs makes it difficult to demonstrate concepts and constructions by hand. In this respect our visual tool is at its best.

Next in Section 1.1, we take a look on relevant tools and literature. In Section 2, we consider what kind of constructions and concepts could be emphasized in object-oriented programs. In Section 3, we give a short overview of the properties of JavaEmphasizer tool. Future developments are discussed in Section 4.

1.1 Related tools and literature

Although there exists quite many tools for emphasizing parts of programs, there does not seem to be any tool that is intended for learning basic programming concepts. Highlighting tools like `Myer` in Yavner (2003), `C++2HTML` in Bedaux (2003), and `Syntax Highlighting` in Ostermiller (2002) produce highlighted program code (usually HTML), but they support only a small set of constructions. Typically those support about 10 different syntactic features. Those tools are intended for increasing the readability of documentation – their aim does not seem to be in teaching emphasized entities. None of the tools seems to support emphasizing constructions based on some semantic meaning.

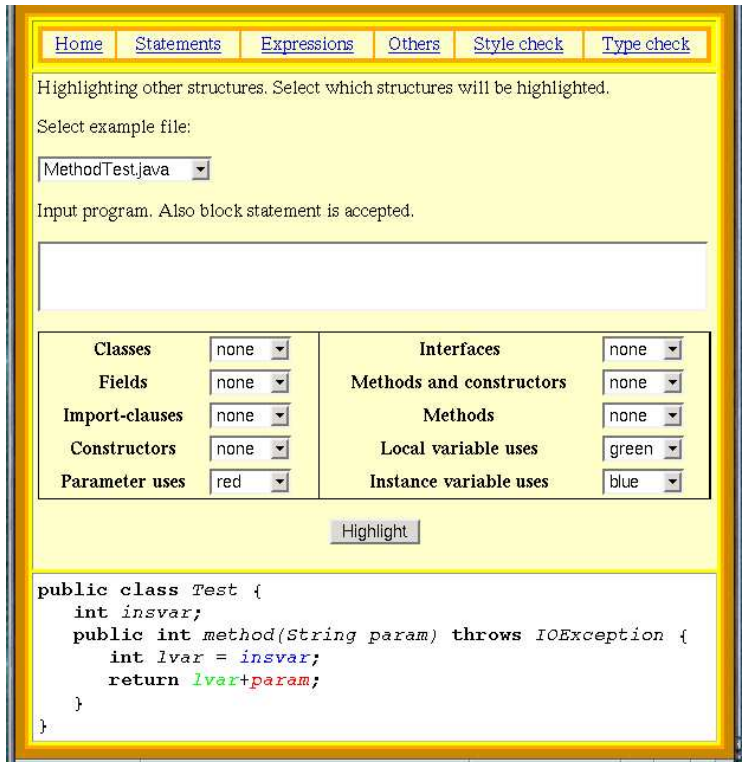


Figure 1: Screenshot of the user interface. The first menu list can be used to select a file from a predefined set of examples. The user can also input a program to the text box.

Surprisingly little appears to have been written about emphasizing program code. There seems to be no studies on what semantic properties could be recognized from programs – though, the topic is in a minor role in Halonen et al. (2004). Using emphasizing, more precisely colors, to increase readability and comprehensibility of programs is studied in Rambally (1986) – that study focuses on psychological issues.

Many of the current basic programming books, e.g. Riley (2003); Savitch (2004), teach programming constructions and concepts by giving rather large illustrated program excerpts. However, those do not come up with an interactive tool for illustrating basic concepts.

2 Emphasizing object-oriented programming constructions and concepts

Next, we consider what language constructions and concepts we can emphasize. In principle, it is possible to recognize almost any kind of properties from programs and emphasize those. In practice, recognizing *syntactic properties* is relatively easy whereas *semantic* or even *pragmatic properties* are harder to identify. In this paper we do not consider pragmatic properties.

It should be clear that recognizing all kind of syntactic elements is straightforward. For example in Java, we can easily identify all reserved words, if-statements, expressions, method calls, different kind of classes (abstract, concrete, interfaces, ...), instance variable declarations, method declarations, creation of objects, string literals, etc. Sometimes one would also like to distinguish one element (or some) among all the matching elements, e.g. 'emphasize formal arguments of method XYZ'. One drawback in this sense is that although recognizing is straightforward, specifying the distinction is not always very easy. At the moment our tool does not support distinction features.

Recognizing syntactic constructions that have some special semantic meaning is much harder, since it usually requires analysis of the program. For example, recognizing declaration of instance variables is easy, but showing all places where instance variables are used requires identifying which identifiers in certain context represent instance variables.

We consider that constructions that are identified due to some special semantic meaning are applications of some *programming (language) concept*. The following list is an attempt to classify such programming concepts.

Identifiers with special meaning. E.g. instance / static / local variables, named constants, formal arguments, method names, class / interface names, (basic) types, and exceptions.

Properties of variables. E.g. scope, initialization, applied as L-value, or R-value.

Expressions based on their type. Object valued, boolean valued, exception valued, etc.

Casting. Explicit and implicit casts of values of both basic types and object types.

Special kind of statements. E.g. unreachable statements, nonterminating loop, forced transfers of control, and increment/decrement of a counter.

Statement sequences. E.g. swapping the value of two variables, and nested loops.

Classification of methods. Ordinary methods are often divided according to three attributes: functionality (procedures vs. functions), visibility (private vs. protected vs. public), and role with respect to class (static vs. instance method). Other classification can also be found (e.g. recursive vs. non-recursive, overloaded vs. non-overloaded, inherited vs. non-inherited).

Classification of classes. E.g. abstract vs. concrete, generic vs. non-generic, methodless class, subclasses, superclasses, exception class, collection, singleton, library class (e.g. Math), and ordinary class representing some (real-life) concept.

Our tool supports the recognition of most of the above mentioned syntactic and semantic properties. Besides those, it can for example show the evaluation order of expressions.

3 Tool for emphasizing Java programs

Our tool, Mäkelä (2004), reads a program (from the web) and highlights it with given instructions. It can also highlight a program from a set of predefined examples. We also have a command line based version of our tool that can be used for producing LaTeX output.

Currently the tool is used via web pages, where the user can set what is emphasized and how. The pages then call the tool with proper parameters. We have created several pages, each focusing on a certain area of syntactic constructions and/or semantical concepts.

Due to space limitations, in the following we can only give a glimpse of the tool properties.

3.1 Parsing and lexing

When a program is loaded into the system, it is first lexed and parsed into an object-oriented representation following the Java 5.0 grammar. The parser generator we chose to use is Java CUP, Hudson (1999). Because we want the output program to be indented the same way as the original, and we also want to save the comments, we need to associate additional information with the program elements. Usually the program representation abstracts from these features.

3.2 Highlighting

The tool supports highlighting based on colors, fonts, and style as well as combinations of them. Various lexical elements have default styles that can be overridden.

After parsing the program into a parse tree, highlighting syntactic elements is very easy. The tree can be simply traversed and highlighting can be added according to the type of the

elements. Implementation of highlighting uses so called filters that can be used to select parts of the elements to highlight or select if the element should be highlighted. This is just a special case of tree traversal. We have also implemented a more generic mechanism for traversing the syntax trees that can be used to add highlighting.

To highlight instance variables or global variables properly we should have information about parent classes available. Currently, we just ignore the variables that are not declared in the given source file.

3.3 Features

The program can highlight all different syntactic elements like classes, interfaces, methods and all kinds of statements and expressions. It is also possible to have more complex syntactic analysis using filters that highlight subparts of the structures. To highlight all expressions that are used as L-values, we just combine the filter to highlight left operands of operator expressions, and the filter to highlight expressions with assignment operators.

There is also a feature for highlighting subelements of statements or expressions. This can be used to illustrate the nesting of statements or the evaluation order of expressions (see the example in Section 3.4).

For elements that have associated modifiers (public, static, etc.), we can highlight based on what kind of modifiers the elements have. Using this feature we can highlight methods and fields based on their visibility, and classes based on whether they are abstract or concrete.

For highlighting the uses of variables, we can highlight the uses of instance variables, local variables, and parameters. It is also possible to generate hyperlinks with unique identifiers for each variable in the program. With this feature it is possible to implement a feature that shows the scope of a selected variable.

Some features, like highlighting uses of static variables in non-static methods would be easy to implement by combining the existing filters, but they are not yet implemented. The mentioned example does not work because highlighting instance variables is applied to classes, not methods. It is also tedious to implement this kind of feature using command line parameters. More complex configuration files are needed.

3.4 An example

The most simple part of the program is to output the program in text, HTML or LaTeX. The tree is just traversed and the elements are printed in order. The highlighting styles are kept in a stack.

Figure 2 has an example output from the program. Eight levels of precedence for expressions have been highlighted with different colors.

The program can be configured to use any HTML or LaTeX tag for highlighting the elements. One useful example is having tooltips for the elements using JavaScript.

4 Discussion

The tool is new and its usefulness needs to be experimentally evaluated. Although we find this tool very useful, we consider using first year students to evaluate the tool this autumn. Perhaps, experimental evaluation will reveal some needs to support certain semantical (basic) concepts.

At the moment, the tool only supports Java. Making similar tools for e.g. C/C#/C++ is rather straightforward. We aim to make such tools.

Perhaps the most interesting future direction is to construct a tool that can identify concepts that cannot be considered elementary. There are a lot of domains for different kinds of applications and corresponding library sets. A contemporary fact in programming is that only a small fraction of what is useful (or even necessary) to learn is in the programming

```
public class ExprTest {
    public static void main(String[] args) {
        // In the initialization, one can use literal expression.
        int a = 1, b, c, d, e, f, g, h;
        // Associativity.
        b = c = d = e = f = g = h = a;
        a = a+b+c+d+e+f+g+h;
        ArrayList<String> asd = new ArrayList<String>();
        boolean q = a < 10 ? TRUE : FALSE;
        // Evaluation order.
        a = b && c + 2 < 10 >>> 3 - 1 * t;
    }
}
```

Figure 2: Example output from the program. Simple expression are shown in blue. In complex expression the color illustrates the evaluation order (red first, blue last).

language – nowadays, more and more are in the libraries – those contain important classes, but there is more: concepts. For example, GUI applications have a lot of concepts of their own: event handlers, GUI components, containers, etc. A future challenge is to design an emphasizing tool that can recognize the concepts related to some domain.

References

Bedaux, J., 2003. C++2HTML: Convert C++ code to syntax highlighted (colored) HTML. Http://www.bedaux.net/cpp2html/.

Halonen, H., Närvänen, K., Mikkonen, T., 2004. Automatic Checking of Symbian Coding Conventions. In: Proceedings, 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004. pp. 79 – 88.

Hudson, S., 1999. CUP Parser Generator for Java. Http://www.cs.princeton.edu/~appel/modern/java/CUP/.

Mäkelä, S., 2004. JavaEmphasizer User Manual, version 1.0. University of Turku.

Ostermiller, S., 2002. Syntax Highlighting. Http://ostermiller.org/syntax/.

Rambally, G., 1986. The Influence of Color on Program Readability and Comprehensibility. In: Proceedings, 17th SIGCSE Technical Symposium on Computer Science Education. ACM Press, pp. 173 – 181.

Riley, D., 2003. The Object of Java, BlueJ Edition. Addison Wesley.

Savitch, W., 2004. Absolute Java, International Edition. Pearson.

Yavner, J., 2003. Myer - Semantic Highlighting for C source. Http://home.comcast.net/~jyavner/myer/.

Japroch: A Tool for Checking Programming Style

Sami Mäkelä and Ville Leppänen
University of Turku and TUCS, Department of Information Technology

sajuma@utu.fi, ville.leppanen@it.utu.fi

Abstract

Following a good programming style is essential basis for making programs. To aid the teaching of Java programming, we present a general style checking tool that visually shows the places in programs, where a chosen style is not followed. Our tool has a configuration language that enables one to describe many kind of good coding conventions.

1 Introduction

Many studies, Mengel and Yerramilli (1999); Oman and Cook (1990); Rees (1982); Truong et al. (2004), emphasize that when teaching programming for the first year students, it is very important to express, what good programming style means. Almost every programmer has a vague understanding on good programming style, but finding a rigid definition is difficult. A notable exception in this respect is 'Java Code Conventions', Sun Microsystems (2004), although it can be argued whether the conventions capture all aspects of good programming style and whether all the values for style parameters are the best possible. To some extent the matter is subjective. However, studies of Berry and Meekings (1985); Mengel and Yerramilli (1999); Miara et al. (1983); Sun Microsystems (2004); Oman and Cook (1990); Rees (1982); Truong et al. (2004) list a similar set of attributes for defining a good programming style. The proposed ideal 'values' for attributes differ though. The idea in many studies has been to develop a metrics to measure the quality of programming style.

In this paper, we describe an interactive tool, *Japroch* (Java programmable checker; see Figure 1), that can be used to check whether a Java 5.0 program is written according to some style. Because there is no consensus on a single good programming style, our tool is very flexible supporting many kinds of style checking aspects via its configuration language. The current version of the tool focuses on typographical issues, as do Miara et al. (1983) and Oman and Cook (1990), but the tool can also reveal some violations related to semantical or logical usage of constructions. Moreover, there is a clear idea (Section 4), how our tool can be generalized to support checking more demanding properties of programs, related to the semantics of certain constructions. Besides providing feedback for the students, one purpose of this tool (as in Jackson (1996)) is to decrease the time spent on checking student programming exercises. For students the tool helps to expose possible style errors in their coding process.

1.1 Related work

Some of the early studies Berry and Meekings (1985); Kernighan and Plauger (1982); Leach (1995); Miara et al. (1983); Rees (1982) involved implementing a checking tool for Pascal or C programs. Some of those studies even considered Cobol and Fortran. Newer studies consider Ada (and C; Jackson (1996)), C++ (Mengel and Yerramilli (1999)), and Java (Truong et al. (2004)).

The tool described in Truong et al. (2004) focuses on finding syntax errors, semantic errors, and logical errors. Some of the errors can be considered as style errors, but mostly their tool operates in a different domain than ours. Mengel and Yerramilli (1999) do not really construct a style analysis tool, but use Verilog Logiscope tool to extract style related parameters from programs. Their main idea is to form a style metrics for C++ programs. An extensive tool for C++ is Style++ Uimonen and Koljonen (2000) – Style++ covers similar aspects as Checkstyle Burn (2003), but for C++ instead of Java. The Ada related tool reported in Jackson (1996) considers correctness, efficiency, complexity, and test data coverage besides the style issues.

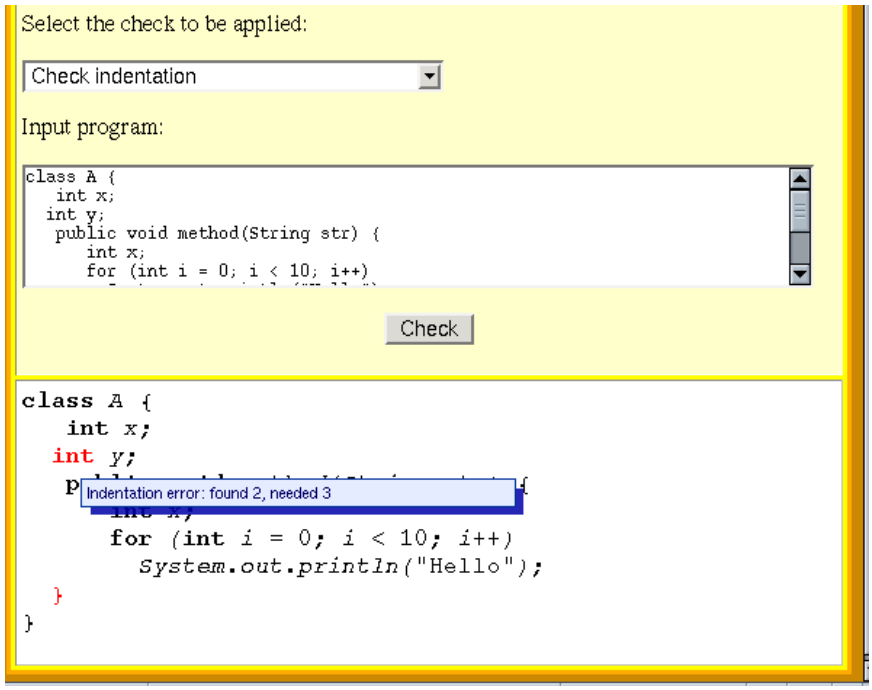


Figure 1: Screenshot of the user interface. The tool is instructed to use a very simple style 'check indentation', which only looks for indentation errors (found errors shown in red).

In the style issues, the tool follows guidelines given by Berry and Meekings (1985). The tool reports numerical values for all the aspects – those could be used to define a metrics for Ada programs. The configurability of the Ada tool is unknown to us.

In Internet distribution, there appears to be one extensive tool, Checkstyle, Burn (2003), for checking the style of Java programs. The tool is formed to match the Java coding conventions provided by Sun Microsystems (2004). Checkstyle has XML-based configuration file, where the user essentially can specify, which predefined checks are applied and with which parameter values (e.g. maximum length of lines, value for indentation). The tool does not seem to allow flexibly construct new kind of checks as our tool does. On the other hand, Checkstyle is currently more extensive than our tool. Checkstyle reports found style errors by giving line numbers – whereas our tool shows them visually. Reporting of Checkstyle can be made more visual by configuring it to work with Eclipse.

2 Elements of good programming style

What are the elements of good programming style? Space limitations prevent us to try to fully answer to the question. Perhaps, the most fundamental work is Oman and Cook (1990), where a taxonomy for good programming style is defined. Their style taxonomy consist of four parts: general programming practices, typographical style, control structure style, and information structure style. With respect to object-oriented languages, the last part seems obsolete (the taxonomy is written with respect to C and Pascal).

Practical studies of Berry and Meekings (1985), Jackson (1996), Kernighan and Plauger (1982), Leach (1995), Sun Microsystems (2004), Miara et al. (1983), Mengel and Yerramilli (1999), Rees (1982), and Truong et al. (2004) do not fully follow the proposed taxonomy, but rather list style topics that are considered. Also the Checkstyle tool, Burn (2003), does not use the taxonomy to classify its features.

We consider that style issues should be classified into four categories:

Typographical. All issues that are related to the visual outlook of the program. These are mainly related to the usage of white space characters. E.g. indentation, placement of

parenthesis, maximum length of line, and method length (in number of statements).

Syntactic. All issues, where some usage of language construction is syntactically correct, but represents bad programming style. E.g. every switch-statement should have a default-branch, and each case-branch should end to a break-statement.

Semantic. This refers to cases, where the semantics of a construction affects the way it should be used. E.g. class names begin with a capital letter and each declared variable should be used in the program.

Logical. Issues related to the logical structure of the program. E.g. there should not be too deeply nested loops, methods should not have a huge number of parameters, and global variables should not be used as method parameters.

Currently our tool supports typographical and syntactic categories well. Some issues belonging to the other two categories can also be checked, but a more extensive support requires modifications to the tool (see Section 4). The recent study of Halonen et al. (2004) presents a tool for checking the semantic style issues of Symbian C++ coding conventions.

3 Style checking tool for Java

We have implemented a tool that can check for several style related issues in Java programs. New checks can be easily implemented using our configuration language. Currently the checker is about 3000 lines of code, and the libraries for parsing and printing are about 20000 lines.

The program is used through a web interface. With it the user can input a program and select which kind of check will be applied to the program. In the web page generated by the program, we have the locations with errors colorized, and when the user moves the mouse over these locations, we show the related error messages, see Figure 1.

3.1 Features

The styles that we check for are implemented as configuration files. The format of these files will be described later. Currently the program can check for simple typographical issues like indentation and whitespaces, and syntactic checks like checking that all switch-statements have default-cases, or checking that some elements are in a desired order.

There are several different ways to indent the program. For example the placement of braces might be different. Our program can enforce various indentation styles, and many more can be defined easily. The tool can also check for whitespaces in for-, if-, do-, and while-statements.

There are different possibilities on how to indent local variable declarations:

Custom indentation:	Default indentation:
`int x = 0,`	`int x = 0,`
`    y = 23;`	`   y = 23;`

Because code editors have automatic indentation, but they cannot be configured for custom format, it is probably advisable style to forbid declaring several different variables in the same statement. We have also implemented this check.

It is a common error to use operator = instead of ==. Because of this problem, we implemented a check that checks that assignment operators are not included inside other expressions. Even if their uses would be correct, those can make the program harder to read.

We have also implemented checks that ensure that some elements are in a desired order. These include checking modifier order, checking that variables are declared in the beginning of a block, and checking that members of a class are defined in a specified order.

Some other checks are checking for default-case in switch-statements, checking that all cases in a switch are terminated by break, checking for length of lines and checking for

redundant boolean literals. It is also possible to check for C-style array declarations in variable and method declarations.

3.2 Configuring the checker

To check for style we need to traverse the parse tree. The inner nodes of the parse tree are syntactic elements of the Java 5.0 programming language. They have a list of subnodes. The leaf nodes are string tokens.

The tree is traversed using the following architecture: A traversing component gets a tree node and an environment as arguments. The component can access the tree, but cannot modify it. The component can access and modify the environment. As a result, checking of the component returns a success, a failure, or a partial failure which means that there was a style error. In addition to these components for nodes, there are similar components for lists of tree nodes. These components are like normal components, but they additionally have a pointer to an element of the list. This enables iterating the elements of the list.

There are several ways to combine components to implement more complex checkers. Boolean operators **and**, **or** and **not** make checks that require both checks to pass, one of the checks to pass or the check to not pass. Iteration structure $*\,c$ applies check c to the elements in the list until all elements are checked or it finds a node that fails the check. Option structure $?\,c$ applies check c to the element in the list, and if succeeds, moves to the next item in the list.

Selection structure **guard**$\{c_1 : a_1 \ldots c_n : a_n\}$ applies checks $c_1 \ldots c_n$ to the node in order. If a check c_i succeeds, to the corresponding action a_i is applied to the token. If none of the checks succeeds, the first partially failed case is selected. The action can add a warning to the checked node, or it can apply more checks to the node.

Figure 2 has an example of the use of the **guard**-construct. It is usually used to traverse the tree recursively, with cases that detect special kinds of elements. The checker defined in the example is given name *root*. It means that it is applied to the top node of the the parse tree. The case **class** *LabeledStatement* checks if the node is of class `LabeledStatement` in the internal representation. When a labeled statement is found, the program outputs a warning. For leaves of the tree there is no check. If the node is an inner node (ie. not leaf), all of its subnodes are traversed recursively to find all labeled statements.

root = **guard**{
  class *LabeledStatement* : **warn** "Labeled statement";
  leaf : **true**; **true** : [$*$ *root*];
}

Figure 2: Example use of the guard-construct.

The environment contains indentation levels in a stack and error messages that are associated with nodes of the parse tree. If a check in the selection structure fails, the changes it made to the environment must be cancelled. Because of this the old environment must be stored. This implies much copying. To make copying efficient we implemented the environment using functional data structures.

For indentation and whitespaces, we can check if the token is indented properly, check if the token has correct number of leading or following whitespaces, and start or stop new indentation level. Indentation levels are stored as a stack in the environment. To handle various possible styles, there are many different kinds of indentation levels, e.g. levels relative to the first token that belong to the level, or levels that are relative to the number of nested braces. There is also a feature that checks a leaf token against a regular expression. This can be used to check that an identifier has a standard prefix or postfix.

4 Discussion

Currently our program supports mainly checking syntactic features. An exciting direction for future research is incorporating semantic checks to our tool. To implement this, we need to associate into the tree nodes type information and perhaps some additional information collected from an overall analysis of the program classes (and of the stardard library classes). That information we could use by introducing a new action into the configuration language that would allow calling arbitrary tree nodes checkers written in Java (using the reflection). Such checkers could be implemented using the design pattern 'command'.

We still need to consider the expressiveness of the current configuration language. For example, it should be expressive enough to express almost all of the Java Code Conventions Sun Microsystems (2004), but this has not been verified yet.

Finally, it should be noted that so far we have not made an extensive experimental evaluation of the usefulness of our tool, but once the discussed extension to the tool is made, the evaluation should take place. At that point, we should also construct definition(s) for good programming style taught at our university.

References

Berry, R., Meekings, B., 1985. A Style Analysis of C Programs. Communications of the ACM 28 (1), 80–88.

Burn, O., 2003. Checkstyle Home Page. Http://checkstyle.sourceforge.net/.

Halonen, H., Närvänen, K., Mikkonen, T., 2004. Automatic Checking of Symbian Coding Conventions. In: Proceedings, 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004. pp. 79 – 88.

Jackson, D., 1996. A Software System for Grading Student Computer Programs. Computers & Education 27 (3/4), 171 – 180.

Kernighan, B., Plauger, P., 1982. The Elements of Programming Style. McGraw-Hill, Inc.

Leach, R., 1995. Using Metrics to Evaluate Student Programs. SIGCSE Bulletin 27 (2), 41–43.

Mengel, S., Yerramilli, V., 1999. A Case Study of the Static Analysis of the Quality of Novice Student Programs. In: Proceedings of 30th SIGCSE Technical Symposium on Computer Science Education. ACM Press, pp. 78–82.

Miara, R., Musselman, J., Navarro, J., Shneiderman, B., 1983. Program Indentation and Comprehensibility. Communications of the ACM 26 (11), 861–867.

Oman, P., Cook, C., 1990. A Taxonomy for Programming Style. In: Proceedings of the 1990 ACM Annual Conference on Cooperation. ACM Press, pp. 244–250.

Rees, M., 1982. Automatic Assessment Aids for Pascal Programs. SIGPLAN Notices 17 (10), 33–42.

Sun Microsystems, 2004. Java Code Conventions. Http://java.sun.com/docs/codeconv/.

Truong, N., Roe, P., Bancroft, P., 2004. Static Analysis of Students' Java Programs. In: Proceedings of Sixth Conference on Australian Computing Education. Australian Computer Society, pp. 317–325.

Uimonen, T., Koljonen, K.-P., 2000. Style++ homepage. Http://www.cs.tut.fi/ style/.

PART 4
Poster presentations

Creation of self tests and exam questions as a learning method

Teemu Kerola and Harri Laine
Department of Computer Science, University of Helsinki

Teemu.Kerola@cs.helsinki.fi

Abstract

Computer checkable self tests and quizzes are a technique used in many e-learning courses. Typically the problems have been defined by the course designer. In this paper we introduce an approach of using students as composers of these problems. We discuss briefly the benefits of this approach and the problems of implementing it using current e-learning environments. We discuss also tool requirements for this approach.

1 Computerized feedback problems

Many web based e-learning courses contain self tests, quizzes or assessment tasks that are marked by computer. They are mainly used by the student herself to assess her skills in the topic at hand. The problems may also be used to set up an exam. These tasks are interactive. The student responds to a question and the system instantaneously marks it and gives a short feedback about the answer. Feedback for an incorrect answer should be educational and instruct the student to find a correct answer. Feedback for correct answer may offer some new insights to the problem.

Learning environments provide varying types of computerized feedback problems. Single response multiple choice questions are most common. Other types include multiple response multiple choice problems, value range problems, fill in the blanks problems, and ordering and joining problems. More complex dedicated problem types like writing programs or database queries have also been uses in Computer Science courses. The setting of the problem may be graphical or textual. For example, a value range can be given within a picture or graph, or an ordering problem can request the student to assess a set of pictures and put them in some specific order.

Most environments provide only static problems. Dynamic problems contain parameters and offer a larger variety but marking and providing proper feedback is more complicated.

2 Students as authors of computerized feedback problems

The usual way of students to get involved in computerized feedback problems is that the teacher has included them in the course material. Some learning environments keep track of students progress, but often problems can be solved anonymously.

Another, more challenging way is to let students to author the problems for other students. This implements the learning by teaching paradigm (Skinner, 1993) (R. Ploetzner, 1999). To author a new problem one must understand the topic well enough to use the question to teach other students something. One must create a relevant question, determine the correct answers for it, consider what are the typical faults, and determine what are the proper feedbacks for correct and incorrect answers.

Authoring new questions is a task very suitable for co-operative work. Students can first discuss the topic area to determine good questions that are not yet included in the course material. Finding out potential answers and proper feedbacks for them is even more challenging, and allows the students naturally to have more discussions on the topic and the pedagogy to teach it.

3 Implementation of problem authoring and problem evaluation

We have used web based self tests in many curses but self test authoring by students only in two courses. We found that implementing this approach using the current e-learning

environments was hard. Many learning environments provide a good teacher interface for authoring self tests and quizzes. They also provide easy ways for students to use the questions. However, they cannot be easily used by students to author the problems. There is typically too much administrative work and too little feedback on the use of the tasks. We would like student teams to author problems and then publish them to other students for evaluation. For example, in WebCT this would require the system administrator to establish a separate course for each student team. Students should be defined as instructors for these courses. Instructors, however, cannot test the tasks so the students would need another identity for that. WebCT has detailed student activity tracking, but when self-tests are concerned, no statistics are provided. Also, transferring questions from one course to another is cumbersome with standard instructor tools.

An ideal tool to support this approach would provide a simple way for students to build up course specific teams with minimum instructor effort. It should provide a large variety of computerized feedback problem types and simple user interfaces for composing the questions and defining their correct and incorrect answers as well as their feedback. The members of the authoring team should be able to try out the problems first by themselves and then publish them for peer evaluation. The tool should provide statistics on how the questions are answered, but in addition it should provide an evaluation mode that would provide an evaluation form for the student to fill when she is done with the problem. This form should contain questions about how easy, useful, laborious, educational, and essential the problem was. It should also contain questions about the quality of the feedback provided. An overview of the evaluation data should be automatically included in task specific metadata. These could be used later on, for example, in selecting tasks for an exam. The tool should store the tasks in standard format (IMS, 2004).

We are currently building a prototype of the tool outlined above.

References

IMS, 2004. Ims Question and Test Interoperability Specification, Version 2.0, public draft.
URL http://www.imsglobal.org/question/index.cfm

R. Ploetzner, P. Dillenbourg, M. P. . D. T., 1999. Learning by explaining to oneself and to others. In: Dillenbourg, P. (Ed.), Collaborative-learning: Cognitive and Computational Approaches. Elsevier, pp. 103–121.
URL http://tecfa.unige.ch/tecfa/publicat/dil-papers-2/Dil.7.1.12.pdf

Skinner, J., 1993. Learning by teaching at the university. English Teaching Forum Online 31 (4), 40–40.
URL http://exchanges.state.gov/forum/vols/vol31/no4/p40.htm

Guidelines for Reducing Language Bias in the Computing Sciences: Language Lessons Learned from a Sister Science

Justus Randolph
Department of Computer Science
University of Joensuu

justus.randolph@joensuu.fi

Abstract

This article describes the need for researchers and practitioners in the computing sciences to have codified guidelines for the responsible use of language. Guidelines from the American Psychological Association concerning language related to gender, age, sexual orientation, disability, and racial and ethnic identity are summarized.

1 Introduction

Having worked as a language reviewer in the computing sciences for some time now, it has become clear to me that there is a need for unified writing guidelines in the field, especially when it comes to dealing with language bias. Many of the papers I review have language bias flaws; however, to the credit of the authors of these papers, they clearly are attempting to write without bias. The mistakes they make are systematic and illustrate an effort to use culturally competent language. For example, in a first draft of a thesis that I recently reviewed, the author wrote

> By combining *her* knowledge about the birds flying, the physics of aerodynamics, and the engineering skills of construction and propulsion, *a man can overcome her natural limitations* and fly with the help of an airplane. (italics mine, reprinted with permission)

The use of *her*, twice, to refer to a *man* shows sensitivity and even a strong commitment to using nonbiased language, even to the point of absurdity. Because I have seen similar examples of this many times, I hypothesize that the cause of this problem is not a lack of commitment to reducing language bias in the computing sciences. Rather, I suggest that these language bias errors are the result of a lack of codified rules for language use.

If there were guidelines for responsible language use in the computer sciences, it is reasonable to assume that they would be included in the ACM digital library (The Association for Computing Machinery, 2004). However, a search of the ACM digital library using the keywords *style guide*, *publication manual*, and *language bias* yielded no search results that led to a unified style guide or publication.

If we are to be ethical practitioners and researchers in the computing sciences, we are obligated to be aware and responsible language users. We should ensure that we use language that is not biased in terms of gender, sexual orientation, disability, or racial and ethnic identities. However, since the rules governing the responsible use of language in the tradition of the computer sciences are at best implicit, it might benefit us to adopt those rules from other sciences, particularly the social sciences, which have already cut their teeth developing the rules for the responsible use of language.

In my opinion, the most reasonable solution to the problem, at least for the time being, is to adopt the guidelines that the American Psychological Association (APA) uses for reducing language bias. The APA publication manual (American Psychological Association, 2001) is now the accepted style guide for over 1,000 journals in a number of fields including psychology, the behavioral sciences, nursing, and personnel administration. In terms of reducing language bias, it has a well-established set of rules that emphasize sensitivity, specificity, and acknowledgement of participation.

I acknowledge that adopting APA guidelines, because they inherently promote U.S. scientific values and the U.S. variety of English, is in some way replacing one set of biases for another. However, with all things considered, the added value of adopting guidelines for reducing language bias that are of U.S. origin, over having no standards at all, seems to be well worth the effort, at least until a set of field-specific guidelines are formalized.

The rest of this paper gives an overview of the guidelines for reducing language bias that are presented in the fifth edition of the *Publication Manual of the American Psychological Association*. With hope, adopting these guidelines will enable us to share a common voice and solidify our commitment to culturally competent communication.

2 Gender Bias in Language

Guidelines for reducing gender bias in language include avoiding gendered pronouns, nouns, and verbs and using parallel forms in languages.

2.1 Gendered Pronouns

Avoid using a gendered pronoun (e.g., *he, him, his, himself*; *she, her, her(s), herself*) or a gendered pronoun combination (e.g., *he/she, him/her, his/her(s), himself/herself*) when the pronoun can logically refer to either gender. It is best to revise these types of biased sentence so that the gendered pronoun can be replaced by a plural pronoun such as *they*, *them*, *their(s)* *themselves* or so that the gendered pronoun can be avoided altogether. For example, instead of writing

> *The user should then enter his password at this time.*

one should write

> *The user should then enter the password at this time.*

or

> *Users should then enter their passwords at this time.*

2.2 Gendered Nouns, Verbs, and Adjectives

Gendered nouns and verbs should be avoided. For example the whole class of *man*-based nouns and verbs, such as *mankind*, *manpower*, *man-machine interface*, and *to man*, should not be used. Some appropriate alternatives for these words, respectively, are *humankind*, *workforce*, and *user-system interface*. Additionally, one should be cautious when using gendered professional labels. For example, it is appropriate to use *police officer* rather than *policeman*. Avoid using *males* and *females* as nouns unless referring to a broad range of ages. Instead, use *male* and *female* as adjectives or use more specific nouns such as *boy*, *girl*, *man*, or *woman*.

3 Age Bias in Language

Use *boy* or *girl* to refer to people who are high school age or younger. For people over 18 years old, *men* or *women* is appropriate. Avoid using *the elderly* as a noun or *elderly* as an adjective. Use the phrases *older people* or *older* instead.

4 Sexual Orientation in Language

Instead of *homosexuals* use *gay men* or *lesbians*, depending on the context. Use *heterosexual* and *bisexual* as adjectives instead of using *heterosexuals* and *bisexuals* as nouns.

5 Disability in Language

Put the person first and the disability second. For example, write *children with disabilities* instead of *disabled children*. Also, avoid words with negative associations such as *victim* or *suffer* as used in *trauma victim* or *a person who suffered a stroke*.

6 Racial and Ethnic Identity in Language

Since preferred designations for racial and ethnic identities are complex and change often, one should ask stakeholders how they prefer to be referred to. In the United States, the currently preferred designations for the largest racial and ethnic groups are European American or White; African American or Black; Hispanic, Latino, or Chicano; Asian, Asian American, or Pacific Asian American; and American Indian or Native American.

7 Conclusion

By adopting these guidelines borrowed from a sister science we can become more responsible language users and, thus, we can also become more ethical researchers and practitioners in the computing sciences.

References

American Psychological Association, 2001. The Publication Manual of the American Psychological Association. Washington DC, 5th Edition.

The Association for Computing Machinery, 2004. The ACM Digital Library. WWW-page, `http://portal.acm.org/df.cfm` (Accessed 2004-08-26).

Author Index