Koli Calling 2009

9th International Conference on Computing Education Research

Arnold Pears and Carsten Schulte

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

Technical report 2010-027
November 2010
ISSN 1404-3203

From the Conference Chairs

This is the proceedings of the 9th Koli Calling International Conference on Computing Education Research.. Once again the unique atmosphere of the Koli Nature Park and the unique conference it hosts for Computing Educators each year has gathered researchers from many corners of the globe for stimulating discussions and presentations. The 9th conference in the series is again run in cooperation with ACM SIGCSE and we thank them for their support.

There are many people to thank, and we extend again our sincere thanks to the support team at the University of Joensuu for logistic support, handling registrations, negotiations with the Koli Hotel and the printing of the pre-proceedings. We also wish to acknowledge the sponsorship of the Routledge - Taylor and Francis Group who have contributed to the conference coffers since 2008.

This year Koli Calling received a total of 29 paper and tool submissions to the conference. After reviewing and due consideration by the programme committee and conference chairs the final proceedings comprises 6 research papers, 12 discussion papers, 2 tools and 2 posters. We hope you enjoy reading these contributions and gain inspiration to develop ideas further and innovate in your teaching and research.

It has been our pleasure to coordinate this conference, and also manage the reviewing of papers and the production of the programme and the proceedings. We hope that you also derive enjoyment and inspiration from the fruit of our labours.

Happy reading.

Arnold Pears and Carsten Schulte
Koli Calling 2009 Conference Chairs

Table of Contents

Session 5. Research Papers 2

Session 6. Discussion Papers 3

Session 7. Poster

Using Roles of Variables in Algorithm Recognition

Ahmad Taherkhani*
Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
ahmad@cs.hut.fi

Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
lma@cs.hut.fi

Ari Korhonen
Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
archie@cs.hut.fi

ABSTRACT

Automatic assessment tools are widely used in programming education to provide feedback for students on large courses and to reduce teachers' grading workload. Current automatic assessment methods typically support analysis of correct functionality and structure of the target program and programming style. Additional features supported by some tools include analysis of the use of specific language structures and program run time. However, no tools have provided a method to check what algorithm the student has used, and give feedback on that.

In this paper, we present a method for automatic algorithm recognition from Java source code. The method falls under the program comprehension research field. It combines static analysis of program code including various statistics of language constructs and analysis of *Roles of Variables* in the target program.

We have successfully applied the method in a prototype for recognition of sorting algorithms although the current method is still sensitive to changes made to recognizable algorithm. Based on the promising results, however, we believe that the method can be further developed into a valuable addition to the future automatic assessment tools, which will have significance on programming and algorithms courses.

Keywords

Static program analysis, algorithm recognition, program comprehension, program understanding, sorting algorithms, roles of variables

1. INTRODUCTION

Algorithms form the basic building blocks of computer science. They have an important role in programming education after CS1, though the first programming courses generally introduce some basic algorithms, like linear search and selection sort. Further courses, CS2 and data structures, on the other hand, often include many programming assignments where students are required to implement or apply specific algorithms to solve problems.

As the first programming courses are often large, many teachers apply some automatic assessment tools, like Boss[21], CourseMarker[18] or WebCAT[11] to provide quick feedback for students, and to reduce their own workload in grading. For an overview of the field, see the survey by Ala-Mutka[1]. These tools are able to analyze many different aspects of the target program, such as program correctness, programming style, program structure, use of specific language constructs, or even run time efficiency. WebCAT introduced a novel aspect to the field, as it supports analyzing how well the program has been tested.

However, none of the existing tools is able to analyze how the problem has been solved in terms of used algorithms. It is difficult to automatically check a programming assignment such as "Write a program that sorts an array using Quicksort that switches to Insertion sort when the sorted area is less than 10 items". The final output of the program is a sorted array and gives no clue what algorithm has been used in reality. A simple approach would be to check some intermediate states, but this is clumsy and unreliable as students may very well implement the basic algorithm in slightly different ways, for example, by taking the pivot item from the left or right end in Quicksort.

This is the starting point of this research. Our aim is to develop methods that could automatically recognize algorithms from source code. This is a challenging problem in computer science, which has similarities to some other problems in the field of *Program Comprehension* (PC). Despite the extensive studies and efforts already done, there still seems to be a lack of an adequate and efficient technique that is able to tackle our research problem.

In general, PC is to find pieces of source code that can be identified to be a particular algorithm. By doing this and viewing them as a whole, we can acquire a partial understanding of the meaning of the code. We call this *Algorithm Recognition* (AR).

In addition to automatic assessment, there are several other applications that could apply the methods and techniques developed to solve the AR problem. First, AR can be used to enable source code optimization, i.e., tuning existing algorithms or replacing them with more efficient ones.

*Corresponding author.

Koli Calling '09, October 29-November 1, 2009, Koli, Finland

This is a key problem in developing compilers for parallel processing machines, that is how to identify algorithms that can be parallelized, and how to replace them with new parallel algorithms that compute the same results. A good overview of this challenging area is presented by Metzger and Wen [24]. Second, many businesses and organizations are faced with the problem of maintaining and developing large legacy codes with documentation that is insufficient, outdated or simply non-existent. This maintenance problem is often addressed with automatic methods and tools that analyze the source code to extract and present information about it on some higher level of abstraction. Much of the previous research in this area has dealt with the recognition of code structures and dependencies such as automatic generation of UML diagrams or dependency graphs from source code. Another problem that has been addressed is code refactoring by identifying clones in the code, that is, pieces of code that implement the same thing [4, 22]. For software maintenance tasks, such information would be very valuable. It helps understanding both the purpose and structure of the code, and provides optimization hints for algorithm replacement. Third, Algorithm recognition techniques can also be used in source to source translations. A widely recognized and employed technique is program translation via abstraction and reimplementation [37], which was introduced to address the weaknesses of the other well-known approach called source to source translation by transliteration and refinement. Finally, it is beneficial to be able to identify clones from larger student programs as well, because recognizing and understanding abstractions is a central skill in building programming competences.

In this paper, we present a new method to recognize algorithms. We describe a prototype that is implemented based on applying static analysis of program code, including various statistics of language constructs and especially roles of variables. These allow us to distinguish between various characteristics of different algorithms. The prototype has been built to distinguish between several common sorting algorithms, but the technique in general can be applied to other classes of algorithms, as well. We emphasize here that the focus of the paper is in the method, not in presenting extensive results of data analysis using the method. Such work remains for future work. An early version of this work has been published in [36].

We start by giving an overview of program comprehension in Section 2 followed by defining the algorithm recognition problem in Section 3. In Section 4, we present a survey of various techniques that have been applied in PC research. Section 5 includes the description of the new method, and in Section 6 we explain how it has been applied in recognizing sorting algorithms. Some initial results are presented. The paper ends in discussion and some conclusions.

2. PROGRAM COMPREHENSION

Program comprehension (PC), also known as program understanding, can be described as a collection of activities related to understanding computer programs, including their functionalities, structures, implementation styles, as well as research on human mental models in the understanding process. Different activities provide different outcomes: knowledge about what the program does, how different parts of the program are related and depend on each others, how the program is implemented, and how it is understood by a human. Basically these outcomes turn low level information into higher level such that can be used in different applications and be further analyzed for different purposes.

In this section, we give an overview of the methods that can be used to tackle the PC problem. We also present a classification of different approaches to the problem.

2.1 Methods applied in PC

Methods in the PC field can roughly be divided into two main categories: *dynamic methods* and *static methods*.

Dynamic methods involve executing the program using some predefined input and examining the output in order to understand the behavior of the program. As the behaviour of a program depends on its input, different kinds of inputs must be used to fully understand the behavior of the program. Although the output for a particular input is always exact, outlining a comprehensive behavior of a program cannot be guaranteed by using dynamic methods. This follows from the fact that the input determines which path of the program will be executed, thus finding a set of inputs that executes all possible paths is difficult.

Despite the shortcomings, dynamic methods provide an indispensable way to ensure that a program works correctly in the intended way. This is perhaps the reason why these methods are mainly used in automatic assessment of students' work. The correctness of the submissions are tested by running their programs using some predefined test inputs and comparing the outputs with the expected values (see for example [10, 18, 21]). Although the dynamic methods are not widely adopted in PC, they can play an invaluable complementary role in solving the problem by bringing in the estimate about the correctness of the code.

Static methods, on the other hand, do not include program execution, but are based on the overall examination of code. In static methods, structure of program is analyzed in a general manner, that is, in a sense of using all the possible inputs. Static methods include many different techniques that can be used depending on the focus of the study. These techniques include analyzing different features of the code such as control and data flow, the complexity of the program in terms of different metrics, and so on. The thoroughness and comprehensiveness make static methods suitable for the PC problem. In fact, most of the PC studies are based on static methods.

2.2 Classification of PC

Based on the objectives and applications, the problem of PC can be classified into the following three categories.

Understanding functionality: The objective of a considerable part of studies in the PC field has been to understand the functionality of programs, i.e., to understand what the programs do. The earlier studies in PC were mainly motivated by the need of software developers, testers and maintainers to understand code without having to read it, which is a time-consuming and error-prone task [17, 27]. An automatic PC tool could be useful in software projects, for example, in verification and validation tasks.

Analysing structure and style: Another category is studying a program from the structural perspective. PC can be seen as examination of the source code, for example, to see how control structures are used and to investigate coding style. The objectives of these analyses could be to monitor students' progress, to ensure that students' submissions are

in accordance with teachers' instructions, and to get a rough idea about the efficiency of the code. Tools that perform these kinds of analyses are mostly used in computer science-related courses at universities and are often integrated into plagiarism detection systems [12, 31, 32].

Discovering programmers' mental model: The focus of these studies is to understand the mental models that represent programmers' understanding of programs. By discovering how programmers understand programs, suitable tools can be developed to support the process of understanding. Although the ultimate objective is to help programmers and maintainers in their work, these studies approach PC mainly from the pedagogical point of view (see for example [35]).

Recognizing and classifying algorithms: PC can also be viewed as the problem of *Algorithm Recognition* (AR). Being able to recognize and classify algorithms implies understanding a program. Therefore, finding out what family of algorithms a given algorithm belongs to or what kind of algorithms it resembles involves PC. The applications of such AR tools include source code optimization, helping programmers and maintainers in their work, examining and grading students' submissions, and so on.

3. ALGORITHM RECOGNITION

We define Algorithm Recognition (AR) to be an activity to outline algorithms from source code. We limit the goal to analyze the source code automatically to identify pieces of code that can be matched against a set of known algorithms. Thus, the aim is to abstract the purpose of the code, and recognize, for example, those parts that have potential for improvement. AR problem thus belongs to the PC research field and can be regarded as a subfield of PC.

There is a close relationship between AR and the activity of PC that studies the functionality of programs. Recognizing an algorithm enables us to discover its functionality. Moreover, by understanding the functionality and implementation pattern of an algorithm, we can determine which algorithm it is. That is, examining the functionality of a given sorting algorithm by matching it against the plans in the knowledge-base system, it can be recognized to be, for example, the Bubble sort algorithm (see [17]).

The same computational task, such as sorting an array or finding the minimum spanning tree of a graph, can be computed using different algorithms. For example, the sorting problem can be solved by using Bubble sort, but also by QuickSort, MergeSort or Insertion Sort, among many others. However, the problem of recognizing the applied algorithm has several complications. First, while essentially being the same algorithm, Quicksort, as an example, can be implemented in several considerably different ways. Each implementation, however, matches the same basic idea (partition of array of values followed by recursive execution of the algorithm for both partitions), but they differ in lower level details (such as partitioning, pivot item selection method or use of recursion). Moreover, each of these variants can be coded in several different ways, for instance, using different loops, initializations, conditional expressions, and so on.

The exact definition of AR problem that we tackle is the following:

AR problem *Let P be a set of algorithms. Given an arbitrary source code S written in a programming language, identify all the implementations of algorithms A_i within S such that each A_i matches some $P_i \in P$.*

Although the aforementioned variation makes AR a difficult task, there are other complexities that make the problem even more challenging. In real-world programs, algorithms are not "pure algorithm code" as in textbook examples. They include calls to other functions, processing of application data and other activities related to the domain, which greatly adds complexity to the recognition process. The implementation may include calls to other methods or the other functionalities may be inlined within the code.

In terms of computational complexity, AR can be regarded to be comparable to many undecidable problems. For example, it is similar to the problem of deciding the equivalency of syntactical definitions of programming languages, which is also known as the equivalency problem of context-free grammars, and is proven undecidable [3]. As will be described in Section 5, we will approach the problem by converting it into the problem of examining the characteristics of algorithms. Furthermore, we will limit the scope of our work to include a particular group of algorithms. In addition, we are not looking for perfect matching, but aim at developing a method that provides statistically reasonable matching results.

4. RELATED WORK

Knowledge-based techniques are one of the earliest techniques adopted to solve the PC problem. These techniques concentrate on discovering the functionality of a program and are based on a knowledge base that stores predefined plans. Plans, also known as idioms, clichés, etc., are frequently used stereotype schemas and are extracted from the algorithms that are meant to be recognized by a knowledge-based PC tool. To understand a program, program code is matched against the plans. If there is a match, then we can say what the program does, since we know what the matched plans do.

As mentioned previously, there is a close relationship between AR and understanding the functionality of a program. Because the latter task is often carried out using knowledge-based techniques, and since these techniques constitute the most widely applied methods in the problem, we give a brief overview of them.

Knowledge-based PC is often referred to as the cognitive process of program understanding, suggesting that the process of understanding a program is about understanding the goal of that program, i.e., the intention of the programmer. This intention is achieved using plans, that is, the techniques used for implementing the intention [33, 28]. In this context, writing a new program is a process of rewriting the goal of that program into a set of subgoals using plans as the rules of the rewriting process. A program can be created by composing simple plans in a complex way. PC can be regarded as a reverse process: understanding the goal through understanding the subgoals using plans. In this reverse process, single plans that are recognized from the target program are combined in a hierarchical manner into plans with higher-level abstraction. The final goal of the target program is ultimately recognized by continuing this process. Heuristics and artificial intelligent techniques are often exploited in the process of concluding the goal of the program.

Algorithms are formed from plans. For example, as Letovsky and Soloway describe [33], Mergesort can be thought of as a collection of the following plans: a plan for recursion on a binary tree, a plan for splitting a sequence into two, a plan for sorting pairs of numbers, and finally a plan for merg-

ing sorted lists. As this example illustrates, plans are not to be confused with algorithms or procedures since they are conceptually different.

Depending on whether the recognition of the program starts with matching the higher-level or lower-level plans first, knowledge-based techniques can be further divided *bottom-up*, *top-down*, and *hybrid* techniques. Most knowledge-based techniques work bottom-up (see, e.g., [17]), in which we try to recognize and understand small pieces of code, i.e., basic plans first. After recognizing the basic plans, we can continue the process of recognizing and understanding higher-level plans by connecting the meanings of these already recognized basic plans and by reasoning what problem the combination of basic plans tries to solve. By continuing this, we can finally try to conclude what the source code does as a whole. In top-down techniques, the idea is that by knowing the domain of our problem, we can select the right plans from the library that solve that particular problem and then compare the source code with these plans. If there is a match between the source code and library plans, we can answer the question of what the program does. Since we have to know the domain, these techniques require the specification of the problem (see, for example, [19]). Hybrid techniques (see, e.g., [29]) use both techniques.

Knowledge-based techniques have been criticized for being able to process only toy programs. For each piece of code to be understood, there must be a plan in the plan library that recognizes it. This implies that the more comprehensive a PC tool is desired to be, the more plans must be added into the library. But, the more plans there are in the library, the more costly and inefficient the process of searching and matching will get. To address these issues of scalability and inefficiency, various improvements to these techniques have been suggested including *fuzzy reasoning* [9].

The idea of applying fuzzy reasoning in PC is that instead of performing the exhaustive and costly task of comparing the code to all plans, a set of more promising pieces of code, i.e., candidate chunks can be selected and the more detailed matching can be carried out only between these candidate chunks and the corresponding plans. The process is as follows. In the beginning, potential chunks are identified from the source code. Chunks are pieces of code that can be understood apart from other parts of code, and do something meaningful independently. For example, a swap operation can be considered as a chunk. The number of chunks that can be identified in large programs can obviously be large. Thus, the number of candidate chunks needs to be reduced and only the most promising chunks should be selected for further examination. This is carried out using some heuristics resulting in a smaller group of chunks, called candidate chunks [8]. Once the candidate chunks are found, the system retrieves from the plan library only those plans that look similar to these chunks. It applies fuzzy reasoning using distinguishing code characteristics to rank these plans according to their similarity to the code and selects the highest-ranked plans to be more closely investigated and compared. Because the more detailed investigation and comparison includes computationally more expensive operations, the efficiency comes from the fact that only selected plans are compared to candidate chunks.

Program similarity evaluation techniques, i.e., plagiarism detection techniques are used to determine to what extent two given programs are the same. The main motivation for these studies and the main application for these tools have been to prevent students to copy each other's works. These techniques focus on the structural analysis and the style of a program, rather than discovering its functionality. Therefore, they fall into the second class of our previously presented classification of PC. These techniques perform many different static analyses on programs including structural analysis, control flow analysis and data flow analysis, among others. This makes them an attractive and useful method in PC. There are several common characteristics of source code used in our method and in these techniques including various complexity metrics (see Section 5). In what follows, we explain these techniques briefly.

Based on how programs are analyzed, these techniques can be divided into two categories: *attribute-counting* techniques [12, 31] and *structure-based* techniques [32]. In attribute-counting techniques, some distinguishing characteristics of the subject program code are counted and analyzed to find the similarity between the two programs, whereas in structure-based techniques the answer is sought by examining the structure of the code.

The most common and distinguishing characteristics that have been used in attribute-counting methods are Halstead's metrics [16]. These characteristics are counted from the program code, and the result is compared with the corresponding result obtained from the other program code following the same procedure. Finally, a number indicating the similarity between two programs is derived from the results by applying some formula, which varies in different works.

The accuracy and reliability of attribute-counting methods have been criticized claiming that these methods fail to detect the similar programs that have been modified even in textual manner. Structure-based methods have been suggested as an improvement, arguing that these methods are much more tolerant to different modifications imposed by students to make the program look different [25].

Structure-based methods can be further divided into string matching based systems and tree matching based systems. As described in [25], string matching based systems use different string matching methods, such as Running-Karp-Rabin and Greedy-String-Tiling (see for example [38]) and parameterized matching algorithm (see, e.g., [2]).

Similarity evaluation techniques have been studied both for analyzing natural languages and program code. Detecting similarities in natural language is generally considered to be more difficult, for example, because of ambiguity and complexity of natural languages. However, some tools are capable of detecting similarities between both programs and natural languages [25].

Reverse engineering techniques are used to understand a system in order to recover its high-level design plans, create high-level documentation for it, rebuild it, extend its functionality, fix its faults, enhance its functions and so forth. By extracting the desired information out of complex systems, reverse engineering techniques provide software maintainers a way to understand complex systems, thus making maintenance tasks easier. Understanding a program in this sense refer to extracting information about structure of the program, including control and data flow and data structures, rather than understanding its functionality. Different reports that can be generated by carrying out these analyses indeed help maintainers to gain a better understanding of program enabling them to modify the

program in a much more efficient way, but do not provide them with direct and concise information about what the program does or what algorithm is in question. Reverse engineering techniques have been criticized for the fact that they are not able to perform the task of PC and deriving abstract specifications from source code automatically, but they rather generate documentation that can help humans to complete these tasks [30].

Since providing abstract specifications and creating documentation from source code are the main outcomes of reverse engineering techniques, these techniques can be regarded as analysis methods of system structure rather than understanding its functionality. Thus their relevance to our method is not high.

Clone detection techniques: aim at finding clones in source code. Clone means the duplication of some piece of a source code, which is either intentionally copied by a programmer from somewhere else in the same system to be reused directly or with some small modifications, or is created by him without awareness of the existence of the same code elsewhere in the same system such that solves the same problem and could have been reused. Clones make the maintenance task more difficult and thus, it is important to find them. The clone detection problem is about searching for the same or almost the same code in a program and in this sense it resembles the plagiarism detection problem.

Clone detection techniques are close to our research, as our purpose is to look for similar patterns of code in the source. However, there is the following essential difference between these two: in clone detection, the goal is to find similar or almost similar pieces of code within a program or software and therefore, all kinds of identifiers that can provide any information in this comparison process can be employed. These identifiers may include comments, relation between the code and other documents, etc. For example, comments may often be cloned along with the piece of code that programmers copy and paste. Our purpose, on the other hand, is to identify implementations of some predefined set of algorithms for human inspection that would support understanding the purpose of the code. We are not looking for similarities within the program. Thus, we can not make use of identifiers like comments in the same way.

Traditionally, clone detection techniques are based on structural analysis, i.e., structural organization, control and data flow of the source code as well as abstract syntax tree analysis (see for example [5]). Marcus and Maletic [22] have introduced a new technique to detect high-level clones in source code. Their method detects clones by identifying the implementation of similar high-level concepts. They use LSI (Latent Semantic Indexing) as an information retrieval technique to statically analyze the software systems and to identify semantic similarity (similar words) among the code. Various files and documentations, as well as comments and identifiers within the source code can be investigated using LSI when trying to find the similarity between two programs and detect clones.

Basit and Jarzabek [4] have designed a tool prototyped named Clone Miner for detecting clones in a file, simple clone as they name it, and clones in different files, which they call structural clones or design-level similarities. Simple clones are detected in a process where source code is first tokenized and then similarities in the token sequence are evaluated. After this, data mining techniques are used to find structural clones. This is carried out by investigating the pattern of co-occurring simple clones in different files. The technique is claimed to be the first one employing data mining techniques to detect design-level similarities, and being capable of scaling up to handle big systems.

In addition to the aforementioned techniques, the following techniques to understand programs or discover similarities between them have also been presented.

Program understanding based on constraints satisfaction [39, 40], *Task oriented program understanding* [13], *Data-centered program understanding* [20], and *Understanding source code evolution* [26].

5. METHOD

Despite the extensive study on PC, much less research has been carried out in the AR field. We introduce a new method for AR that is based on static analysis of source code including various statistics of language. The novelty of our method is the use of roles of variables in AR.

Our approach in recognizing algorithms is based on examining various characteristics of them. By computing the distinguishing characteristics of an algorithm, we can compare these characteristics with those ones collected from already recognized algorithms and conclude if the algorithm falls into the same category.

We divide the characteristics of a program into the following two types: *numerical characteristics* and *descriptive characteristics*. Numerical characteristics are those that can be expressed as positive integers, whereas descriptive characteristics describe some properties of the algorithm in question and are not expressed as numbers. Table 1 shows the numerical characteristics used in our method. The abbreviations in the table are used to refer to the corresponding characteristic in Table 3, which is explained in Section 6. The four last characteristics in Table 1, that is N_1, N_2, n_1, n_2, N and n are the Halstead metrics [16], that are widely used in program similarity evaluation techniques.

In addition to these numerical characteristics, the following characteristics in connection with these characteristics are computed as well:

- *Variable dependencies*: direct and indirect dependencies between each variable that appears in the algorithm to all other variables. If variable X gets its value directly from variable Y, then X is said to be directly dependent on Y. If there is a third variable Z on which Y is dependent (either directly or indirectly), then X is indirectly dependent on Z. A variable may have both a direct and an indirect dependency on another one.
- *Incrementing/decrementing loops*: A loop can be determined as being incrementing or decrementing by examining how the value of its loop counter changes. If the value of a loop counter increases after each iteration, the loop is said to be an incrementing loop and if the value decreases after each iteration, the loop is said to be a decrementing one.
- *Nested/non-nested blocks/loops*: interconnection between blocks and loops. Expresses how many nested/non-nested blocks/loops each block/loop has.

Descriptive characteristics are the following:

- *Recursive*: whether the algorithm is recursive or not.

Table 1: The numerical characteristics used in the method

Abbreviation	Numerical characteristic
NoB	Number of blocks in an algorithm. A block refers to a sequence of statements wrapped in curly braces. A block can be a method or a control structure (loops and conditionals).
NoL	Number of loops in an algorithm. Supported loops are for loop, while loop and do while loop.
NoV	Number of variables in an algorithm.
NAS	Number of assignment statements in an algorithm.
LoC	Lines of code.
MCC	McCabe complexity, i.e., cyclomatic complexity [23].
N_1	Total number of operators in an algorithm.
N_2	Total number of operands in an algorithm.
n_1	Number of unique operators in an algorithm.
n_2	Number of unique operands in an algorithm.
N	Program length ($N = N_1 + N_2$).
n	Program vocabulary ($n = n_1 + n_2$).

- *In-place*: whether the algorithm requires extra memory, e.g., whether it needs auxiliary arrays to carry out the task (used for identifying sorting algorithms)
- *Roles of variables*: what is the role of each variable in the algorithm? This will be explained in more detail in the next subsection.

Using the numerical characteristics, each algorithm can be represented as an n-dimensional feature vector, where n is the number of the numerical characteristics. Algorithms are recognized by identifying their position in the vector space which consists of the set of the algorithms that are supported by the Analyzer (the implemented tool that performs the recognition task based on the method), i.e., the algorithms that exist in the knowledge-base of the Analyzer. Thus, the task of algorithm recognition can be converted to the task of pattern recognition, that is, identifying vectors. In the next steps, a more detailed and accurate process of recognition is carried out by examining the other characteristics related to the numerical characteristics, as well as by investigating the descriptive characteristics.

The recognition process is based on calculating the frequency of occurrences of the numerical characteristics in an algorithm on one hand, and investigating the descriptive characteristics of the algorithm on the other hand. Based on these a decision tree is built to guide the recognition process.

The first phase of setting up the Analyzer is training. Many different versions of the implementation of algorithms that are to be recognized by the Analyzer are analyzed with regard to the aforementioned characteristics and the results are stored in the database. Thereafter the Analyzer has the following information about each algorithm: the type of the algorithm, the descriptive characteristics of the algorithm and the range of values each numerical characteristic can have (within the teaching material). Now, when the Analyzer encounters a new unknown algorithm, it first counts its numerical characteristics and analyzes its descriptive characteristics. Then, the Analyzer retrieves the information of already known algorithms from the database. From this information, the minimum and maximum limits of the numerical characteristics are counted. If the numerical characteristic values of the unknown algorithm fall into the ranges of corresponding minimum and maximum value in the database, the examination process continues. Otherwise, information about the numerical characteristic that is outside this range is given to the user and the algorithm is classified Unknown. Finally, its information is stored in the database.

Due to efficiency, the information of all algorithms should not be retrieved from the database, but the number of retrieved algorithms should be kept as small as possible. This should not, however, cause any relevant algorithm to be left out of the process of matching. Once the unknown algorithm is verified to be within the permitted range, it is examined in more details with regard to its descriptive characteristics. Depending on the results of this examination, the algorithm is assigned a type if it has those characteristics of the same type algorithms existing in the database. Alternatively, if the characteristics of the algorithm cannot be verified as conforming to those of algorithms from the database, it is assigned an "Unknown" type. The information of the algorithm is stored in the database in both cases.

A user can always change the type of an unknown algorithm in the database, if the algorithm can be verified as a particular type by manual inspection. For example, the unknown algorithm might be a Quicksort that is labeled as unknown by the Analyzer because it is longer than the Quicksort algorithm versions existing in the database due to a different implementation style. The knowledge of the Analyzer can be extended in this way. Next time, an algorithm implemented using the same style will be recognized as a Quicksort.

As mentioned, one novelty of the method is the use of roles of variables in the process of recognizing algorithms. In the following, we will present a brief introduction to roles of variables. For more information on roles see [34].

Roles of variables

The concept of role of variables was first introduced by Sajaniemi [34]. The idea behind them is that each variable used in a program plays a particular role that is related to the way it is used. Roles of variables are specific patterns how variables are used in source code and how their values are updated. For example, a variable that is used for storing a value in a program for a short period of time can be assigned a temporary role. As Sajaniemi argues, roles of variables are part of programming knowledge that have remained tacit. Experts and experienced programmers have always been aware of existing variable roles and have used

them, although the concept has never been articulated. Giving an explicit meaning to the concept can make it a valuable tool that can be used in teaching programming to novices, explaining to them the different ways in which variables can be used in a program. Moreover, the concept can offer an effective and unique tool to analyze a program with different purposes. In this work, we have extended the application of roles of variables by applying them in the problem of algorithm recognition.

Roles are cognitive concepts, implying that human inspectors may have a different interpretation of a single variable. However, roles can be analyzed automatically using data flow analysis and machine learning techniques [7, 15].

From all variables in novice-level procedural programs, 99% can be covered using only nine roles [34]. Currently, there are 11 roles recognized that cover all variables in novice-level programs in object-oriented, procedural and functional programming. These roles are presented in Table 2.

An example

Figure 1 shows a typical implementation of Selection sort in Java. There are five variables in the algorithm with the following roles. A loop counter, i.e., a variable of integer type used to control the iterations of a loop is a typical example of a stepper. In the figure, variables i and j have the stepper roles. Variable *min* stores the position of the smallest element found so far from the array and thus, has the most-wanted holder role. A typical example of the temporary role is a variable used in a swap operation. Variable *temp* in the figure demonstrates the temporary role. Finally, data structure *numbers* is an array that has the organizer role.

```
// i and j: steppers, min: most-wanted holder
// temp: temporary, numbers: organizer
for (int i = 0; i < numbers.length-1; i++){
       int min = i;
       for (int j = i+1; j < numbers.length; j++){
              if (numbers[j] < numbers[min]){
                     min = j;
              }
       }
       int temp = numbers[min];
       numbers[min] = numbers[i];
       numbers[i] = temp;
}
```

Figure 1: An example of stepper, temporary, organizer and most-wanted holder roles

6. APPLYING THE METHOD ON SORTING ALGORITHMS

In order to demonstrate the feasibility of our method, we decided to restrict the scope of the work to sorting algorithms and apply the method to the five sorting algorithms Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort. Sorting algorithms were selected because they are widely used, easy to analyze, there are many different algorithms performing the same task and finally, from the perspective of our method, sorting algorithms include both very similar (like Insertion sort and Bubble sort) and yet different algorithms (like Mergesort and Bubble sort). For these reasons, sorting algorithms seemed to be a good choice to test how the method works. In this section, we explain the process of recognizing sorting algorithms.

We manually analyzed many different versions of these common sorting algorithms. Based on the results of these analyses, we posited a hypothesis that the information mentioned in Section 5 could be used to differentiate different sorting algorithms from each other. The problem was whether new unknown code could be identified reliably enough by comparing the information gathered from the unknown code with the information in a database.

As an example of the numerical characteristics, we present the result of analyzing the numerical characteristics of the five algorithms in Table 3. We collected an initial data base containing 51 different versions of the five sorting algorithms for the analysis. All algorithms were gathered from textbooks and course materials available on the WWW. Some of the Insertion sort and Quicksort algorithms were from authentic student submissions. For each characteristic in the table, the first and second number depict, respectively, the minimum and maximum value found from the different implementations of the corresponding algorithm. As can be seen from the table, the algorithms fall into two groups with regard to their numerical characteristics: the small group consists of Bubble sort, Insertion sort and Selection sort, and the big group comprises Quicksort and Mergesort.

We developed a prototype Analyzer that can count all these characteristics automatically. The Analyzer is implemented in Java and the current version is able to process source code written in Java. It parses the code, counts its numerical characteristics shown in Table 3 and analyzes all related characteristics as well as the descriptive characteristics (see Section 5). These information are stored in a database consisting of four tables: *Algorithm*, *Block*, *Variable*, and *Dependency*. A software detecting roles of variables was also integrated into the Analyzer.

Figure 2 shows a decision tree to determine the type of the five aforementioned sorting algorithms. The decision tree was constructed based on the numerical and descriptive characteristics of these algorithms. At the top of the decision tree, we examine whether the algorithm is a recursive one and continue the investigation based on this. Highly distinguishing characteristic like this improves the efficiency, since we do not have to retrieve the information of all algorithms from the database, but only those that are recursive or that are non-recursive. In the next step, the numerical characteristics are used to filter out algorithms that are not within the permitted limits. If a recognizable algorithm has less or more numerical characteristics than those of the other recursive/non-recursive algorithms retrieved from the database, the process of recognition is terminated and an informative error message is given about the particular characteristic that does not fit within the permitted limits.

As can be seen from Figure 2, the roles of variables play an important and distinctive role in the process. In the case of recursive algorithms, Quicksort typically include Temporary role, since it performs swap operation (where the use of Temporary role is typical). This is not the case in Mergesort. Because merging is performed in Mergesort, there is no need for a swap operation there. Thus, a Temporary role does not usually appear in Mergesort.

In the case of the three non-recursive algorithms that we examined, only Selection sort included a Most-wanted Holder. Insertion and Bubble sort do not include selection of a min (or max) element and therefore have no variable appearing in Most-wanted Holder role (for the definition of

Table 2: The roles of variables and their descriptions

Role	Description
Stepper	Assigned to a variable that systematically goes through a succession of values, e.g., values stored in an array.
Temporary	Variable that holds a value for a short period of time appears in temporary role.
Organizer	A data structure holding values that can be rearranged is a typical example of the organizer role. For example, an array to be sorted in sorting algorithms has an organizer role.
Fixed value	A variable has a role of fixed value if it keeps its value throughout the program. The fixed value role can be thought as a final variable in Java which is immutable once it has been assigned a value.
Most-wanted holder	A variable is said to have a most-wanted holder role if it holds a most desirable value that is found so far.
Most-recent holder	Is given to a variable that holds the latest value from a set of values that is being gone through. Moreover, a variable that holds the latest input value is a most-recent holder.
One-way flag	One-way flag is a role assigned to a variable that can have only two values and once its value has been changed, it cannot get its previous value back again.
Follower	Follower is a role indicating a variable that always gets its value from another variable. In other words, its new values are determined by the old values of another variable.
Gatherer	Variable that collects the values of other variables. A typical example is a variable that holds the sum of other variables in a loop and as a result, its value is changed after each execution of the loop.
Container	A data structure into which elements can be added or from which elements can be removed if necessary, has a role of container. For example, all Java data structures that implement Collection interface.
Walker	Can be assigned to a variable that is used for going through or traversing a data structure.

Table 3: The minimum and the maximum of numerical characteristics of five sorting algorithms (see Table 1)

Algorithm	NoB	NoL	NoV	NAS	LoC	MCC	N_1	N_2	n_1	n_2	N	n
Insertion	4/6	2/2	4/5	8/11	13/21	4/6	40/57	47/58	3/6	2/4	87/115	5/10
Selection	5/6	2/2	5/6	10/10	16/25	4/5	47/59	51/57	4/6	2/5	98/116	6/11
Bubble	5/6	2/2	4/5	8/11	15/21	4/5	46/55	49/57	4/6	2/4	95/112	6/10
Quicksort	5/9	1/3	4/7	6/15	31/41	4/10	84/112	77/98	9/17	2/7	161/210	11/24
Mergesort	7/9	2/4	6/8	14/22	33/47	6/8	96/144	94/135	11/14	5/9	190/279	16/23

different roles see Table 2). The rest of the decision making process shown in Figure 2 is self-explanatory.

When testing the method we distinquish the following cases. *True positive* indicates a case where the analyzer correctly recognizes an algorithm that it belongs to the target set of five sorting algorithms. *True negative* correspondingly indicates rejecting an algorithm which is not among the target set. *False positive* denotes that a wrong algorithm is incorrectly recognized as belonging to the target set, and *False negative* correspondingly an algorithm belonging to the set which is not recognized as such.

The Analyzer was tested using various tests designed to produce true positive and true negative cases, as well as false positive and false negative cases. For example, when tested by Shell sort algorithm, the Analyzer correctly gave the following error message: "The algorithm seems to be a non-recursive algorithm that has the following characteristics out of the permitted limit: program length and the number of the loops are above the permitted limit". It is a correct error message, since the Shell sort code was longer and had three for loops.

False negative cases are the most common error the Analyzer makes. Simply by adding some extra and irrelevant

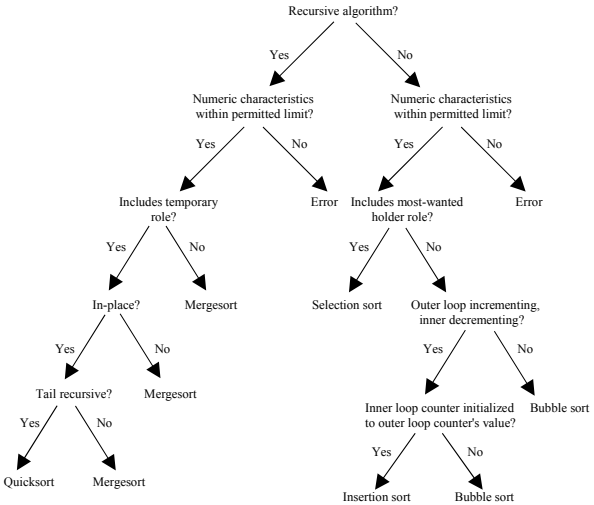


Figure 2: Decision tree for determining the type of a sorting algorithm

code, for example a swap operation or two, the Analyzer can be made believe that the algorithm is not the type that it actually is. These cases, however, result from idiosyncratic code and, as we will discuss in Section 7, our methods is currently not tolerant to irrelevant changes that are intentionally made to the code, and expects algorithms to be implemented in a well-established way.

False positive cases are much rare than false negative, because in order to occur, a false positive case requires that the recognizable algorithm passes the steps shown in the decision tree, which cannot happen so often compared to false negative cases. For example, in order to recognize an algorithm falsely as a Selection sort, the algorithm has to have the right amount of numerical characteristics and additionally, include at least one variable playing Most-wanted Holder. This is not as likely as a true Selection sort being labelled as unknown due to reasons like using an extra variable or failure in producing a correct role (see Section Section 7).

7. DISCUSSION

The Analyzer is a proof-of-concept of the described method, and we have successfully applied it identifying sorting algorithms from tested sample programs. However, the number of tests we have performed is still very small, and there remain important problems to address.

First, we recognize that the method is statistical by nature, and we cannot claim that it could ever achieve 100% accuracy in the recognition problem. Rather, the goal is to minimize the numbers of false positives and negatives. Above we already concluded that false positives are a smaller problem, because the descriptive characteristics and roles of variable give rather detailed and specialized information of the sample code. It is rare that two arbitrary code segments have similar characteristics (both descriptive and numerical, as well as the indicated roles of variables) by chance.

The problem of false negatives is the serious one. Simple additions of application code within the algorithm code, or using slightly different structuring of the code easily lead to the case where the algorithm is not recognized. Therefore, our next step will be applying techniques from knowledge-based program comprehension. We need to identify schemas from the source code and add this to the recognition problem. This would help to match algorithm schemas and identify such schemas which belong to application code. The schema information could be used in building the decision tree, or it could be used as an additional phase for reconsidering cases labeled Unknown.

Roles of variables turned out to be a very distinctive factor that can be used to recognize sorting algorithms. However, they also give rise to the next challenge. Roles of variables are cognitive concepts, which means that different programmers with different background may assign different role to the same variable [6, 14]. To an automatic role analyzer that assigns roles to variables, roles are technical concepts, not cognitive. The challenge rises from making the connection between cognitive and technical concepts. Therefore, it is not easy to develop a role analyzer that can detect the roles with high degree of accuracy (i.e., the roles that are in accordance with programmer's perception or mental model of roles) [14]. However, progress in this field has been made, and the role analyzer we are currently using [7] gives already good results, though we are looking for getting access to even better ones.

The current version of the Analyzer is only capable of classifying algorithms into the five above-mentioned sorting algorithms, or label them as Unknown. We need to extend the Analyzer to other fields of algorithms. This, on the other hand, is very straightforward hard work, which basically only includes training the system with new algorithms and manual analysis of target algorithms to build the more complex decision tree. In addition, the construction of the decision tree needs to be automated.

The current system assumes that the algorithms work correctly. Recognizing incorrect algorithms is out of the scope of the method. Dynamic analysis methods, such that are applied in automatic assessment tools, could be used for that.

Our main motivation is to build a tool that could be used in automatic assessment to provide feedback for students, and as a part of the grading process for the teacher. Here, we have yet a rather long way to go.

8. ACKNOWLEDGMENTS

This work was supported by the Academy of Finland under grant number 111396.

9. REFERENCES

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[3] Y. Bar-Hillel, M. Perles, and E. Shamir. *On Formal Properties of simple Phrase Structure Grammaras.* Zeit. Phonetik, Sprachwiss. Kommunikationsforsch. 14, 1961.

[4] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European Software Engineering Conference*, pages 156–165. ACM, 2005.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pages 368–377, 1998.

[6] M. Ben-Ari and J. Sajaniemi. Roles of variables as seen by CS educators. *SIGCSE Bulletin*, 36(3):52–56, 2004.

[7] C. Bishop and C. G. Johnson. Assessing roles of variables by program analysis. In *Proceedings of 5th Baltic Sea Conference on Computing Education Research, Koli Calling 2005*, 2005.

[8] I. Burnstein, K. Roberson, F. Saner, A. Mirza, and A. Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 102–109. IEEE, 1997.

[9] I. Burnstein and F. Saner. An application of fuzzy reasoning to support automated program comprehension. In *Proceedings of Seventh International Workshop on Program Comprehension, 1999.*, pages 66–73. IEEE, 1999.

[10] S. Edwards. Improving student performance by evaluating how well students test their own programs.

Journal on Educational Resources in Computing, 3(3):1–24, 2003.

[11] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155, New York, NY, USA, 2003. ACM.

[12] B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. In *Journal of Computing Sciences in Colleges*, pages 50–57. ACM, 2007.

[13] A. Erdem, W. L. Johnson, and S. Marsella. Task oriented software understanding. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 230–239. IEEE, 1998.

[14] P. Gerdt and J. Sajaniemi. An approach to automatic detection of variable roles in program animation. In *Proceedings of the Third Program Visualization Workshop (ed. A. Korhonen), Research Report CS-RR-407, Department of Computer Science, University of Warwick, UK*, pages 86–93, 2004.

[15] P. Gerdt and J. Sajaniemi. A web-based service for the automatic detection of roles of variables. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 178–182, New York, NY, USA, 2006. ACM.

[16] M. Halstead. *Elements of Software Science. North Holland, New York*. Elsevier, 1977.

[17] M. Harandi and J. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, January 1990.

[18] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, pages 46–50. ACM Press, 2002.

[19] W. Johnson and S. E. Proust: Knowledge-based program understanding. In *IEEE Transactions on Software Engineering, volume SE-11, Issue 3, March 1985*, pages 267–275. IEEE, 1984.

[20] J. Joiner, W. Tsai, X. Chen, S. Subramanian, J. Sun, and H. Gandamaneni. Data-centered program understanding. In *Proceedings of International Conference on Software Maintenance*, pages 272–281. IEEE, 1994.

[21] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. In *ACM Journal on Educational Resources in Computing, volume 5, number 3, September 2005. Article 2.* ACM, 2005.

[22] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *16th IEEE International Conference on Automated Software Engineering*, pages 107–114. IEEE, 2001.

[23] T. J. McCabe. A complexity measure. In *IEEE Transactions on Software Engineering, volume SE-2, number 4, December 1976*, pages 308–320, 1976.

[24] R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement*. The MIT Press, 2000.

[25] M. Mozgovoy. *Enhancing Computer-Aided Plagiarism Detection*. Doctoral dissertation, University of Joensuu, 2007.

[26] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *ACM SIGSOFT Software Engineering Notes, Volume: 30, Issue: 4, July 2005*, pages 1–5. ACM, 2005.

[27] D. Ourston. Program recognition. In *IEEE Expert, volume: 4, Issue: 4, Winter 1989*, pages 36–49. IEEE, 1989.

[28] S. Paul, A. Prakash, E. Buss, and J. Henshaw. Theories and techniques of program understanding. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

[29] A. Quilici. A memory-based approach to recognizing programming plans. In *Communications of the ACM, volume 37 , Issue 5*, pages 84–93. ACM, 1994.

[30] A. Quilici. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th international conference on Software engineering*, pages 333–336. ACM, 1995.

[31] M. J. Rees. Automatic assessment aids for Pascal programs. *SIGPLAN Notices*, 17(10):33–42, 1982.

[32] S. S. Robinson and M. L. Soffa. An instructional aid for student programs. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, pages 118–129. ACM, 1980.

[33] L. S. and S. E. Delocalized plans and program comprehension. In *Software, IEEE, Volume: 3, Issue: 3*, pages 41–49. IEEE, 1986.

[34] J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 37–39. IEEE Computer Society, 2002.

[35] M.-A. Storey, K. Wongb, and H. Müller. How do program understanding tools affect how programmers understand programs? In *Science of Computer Programming 36 (2000)*, pages 183–207. IEEE, 2000.

[36] A. Taherkhani, L. Malmi, and A. Korhonen. Algorithm recognition by static analysis and its application in students' submissions assessment. In A. Pears and L. Malmi, editors, *Proceedings of Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*, pages 88–91. Uppsala University, 2009.

[37] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14:1207–1228, 1988.

[38] M. J. Wise. Yap3: improved detection of similarities in computer program and other texts. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 130–134. ACM, 1996.

[39] S. Woods and Q. Yang. The program understanding problem: analysis and a heuristic approach. In *18th International Conference on Software Engineering (ICSE'96)*, pages 6–15. IEEE, 1996.

[40] S. G. Woods and Q. Yang. Constraint-based program plan recognition in legacy code. In *Working Notes of the Third Workshop on AI and Software Engineering: Breaking the Toy Mold (AISE-95)*, 1995.

Defects in Concurrent Programming Assignments

Jan Lönnberg
Aalto University
School of Science and Technology
P.O. Box 15400
FI-00076 Aalto, Finland
jlonnber@cs.hut.fi

ABSTRACT

This article describes a study of the defects in the programs students have written as solutions for the programming assignments in a concurrent programming course. I describe the underlying causes of these defects and the applications in developing teaching, grading and debugging of this information.

I present the effects of the students' approaches to constructing and testing programs on their work, how teaching can be and has been improved to support the students in performing these tasks more effectively and how software tools can be designed to support the development, testing and debugging of concurrent programs.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.3 [**Programming techniques**]: Concurrent Programming

Keywords

Concurrent Programming, Defect Cause Analysis

1. INTRODUCTION

An important first step in improving a process is understanding where it fails to produce the desired result and why. Quantitative information is particularly helpful in this endeavour, as it allows accurate and easy prioritisation of possible improvements.

Students' solutions to programming assignments provide information that can be used to improve several interlinked processes. The purpose of an assignment is typically twofold: to allow students to learn to apply in practice what they have been taught and to evaluate how well they have learned. The assignment solutions submitted by students (*submissions*) can also be used to evaluate, indirectly, the teaching the students have received. The submissions can also be used to improve assignments and assignment grading processes to make them determine the students' skills more effectively and accurately.

Information on the defects introduced by programmers can be used as a starting point for the development of debugging methodology and tools. Studying programming assignments in education allows one to get statistically meaningful data on errors made in a specific task. By contrast, in professional development contexts, large numbers of programmers seldom implement the same specification.

In this article, I will describe the defects found in students' programming assignments in a course on concurrent programming and their causes, to the extent they can be deduced. I will then present some conclusions that can be drawn from these data that are relevant to teachers, assignment developers and graders.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

1.1 Related Work

The work described here can be considered to belong to two different areas of research: research on defects in programs and research on students' problems with programming. The former research field aims to improve quality of software by understanding why programmers err, while the latter aims at improving the quality of teaching.

1.1.1 Defects in Software

When studying program *defects* (discrepancies between the actual program and the correct one, commonly known as *bugs*), and the underlying programming *errors* (mistakes), several approaches can be taken that support different approaches to the overarching goal of improving software quality.

One approach, used by e.g. Eisenstadt [8] with the goal of understanding and mitigating difficulties in debugging, is to concentrate on collecting anecdotal data on bugs that were hard to fix and the debugging process involved. The conclusions include types of bugs (such as writing outside allocated memory) that are hard to track down and the methods used by the programmers who tracked them down (e.g. adding `print` statements and hand simulation of execution). Natually, this approach only provides data on bugs that result in a story the programmer finds interesting enough to remember and share.

Another approach, such as that used by Ko and Myers [16] to form an understanding of errors in order to improve error prevention, detection and correction, is to set up an experiment that is videotaped and analysed in detail. This approach can be used to get very detailed information on error causes, especially if the programmers think aloud, allowing their reasoning to be examined in detail. However, this requires a lot of time for analysis (40 h for 15 h of observation), making it prohibitively time-consuming unless one only observes a few students doing a small project. Furthermore, observation of this type is often hard to do in the natural working environment of the students, which may affect their behaviour.

Defects can be classified in a variety of ways, depending on the relevant aspects. For example, Eisenstadt [8] and Grandell et al. [11] form categories based on the observed

defects; others (such as Spohrer and Soloway [26]) construct a classification based on distinctions they wish to study, such as whether the defects are caused by misconceptions about programming language constructs. Defects can be classified, for example, based on their symptoms (how and when the defect manifests itself), or on the difference on the syntactic level between the incorrect code and the intended correct code. In both cases, a variety of different category sets have been formed by different authors.

If sufficiently detailed information is available on the underlying errors, defects can be classified based on the underlying error. Errors can be classified, for example, by the type of cognitive breakdown involved in the error (lack of knowledge, mistake) (e.g. Ko and Myers [16]) or the part of the program design that is incorrect (e.g. Spohrer and Soloway's goal/plan analysis [26]).

In a software development context, many different types of information related to bugs are useful, resulting in a multifaceted classification scheme such as the IEEE standard classification for *software anomalies* (deviations from expectations observed in program operation or documentation, including bugs) [15]. In the IEEE classification, bugs are classified according to a wide range of properties, such as how and when the defect was detected, the type of the defect and the error underlying it and the impact of the defect. Beizer's classification [3] focuses on the aspect of the program or development task that was incorrectly developed, such as requirements, data structures or data processing.

Eisenstadt [8] also classifies by two other aspects that are interesting from a debugging point of view: why the bug was hard to track down and how it was tracked down.

1.1.2 Students' Programming Errors and Misconceptions

Several studies have been made of students' errors in programming assignments (e.g. [11, 26]) and misconceptions about algorithms (e.g. [25]). The goal of these studies is usually to improve programming education by developing an understanding of students' misconceptions and errors.

The programming assignment studies mentioned above all use the students' code as data; either the final versions submitted by the students [11] or every syntactically correct version compiled by the students [26]. This code was then (mostly manually) analysed for defects.

1.2 Applications

As noted in the introduction, information on the types of defects in students' programs can be applied both in developing teaching and grading and in the development of debugging tools and methodology.

1.2.1 Teaching and Assignments

The results of an assignment can be used to determine whether students are effectively learning what they should. In particular, if a large number of students has problems understanding or applying some relevant knowledge, the teaching of this knowledge should be improved.

If students, on the other hand, produce many defects unrelated to the subject matter they are being taught, the assignment may be testing the wrong knowledge and skills. If the defects can be traced to misconceptions about the assignment or the artificial environment in which it is done (if it exists), the students may be distracted from learning relevant matters by difficulties specific to the assignment. Penalising students for defects that are arguably caused by a badly-designed assignment rather than any problem the student may have is hardly just, so it is important to recognise or eliminate these defects.

1.2.2 Code Reviews and Manual Assessment

An experienced code reviewer can quickly spot common defects in the programs he reviews, as he knows what to look for. This applies even more strongly to a grader who reads many similar programs. Information on common defects can therefore be very useful to new graders in a course as a substitute for actual experience (both general and assignment-specific). Information on the errors underlying a defect can be used to guess the error made even in the absence of explanatory reports or comments.

1.2.3 Verification and Automatic Assessment

One of the simplest and most commonly used ways to detect defects in a program is to test it and hope that the defects cause *failures* (incorrect program behaviour). However, testing is practically never exhaustive for non-trivial programs [6] and often gives little indication of the actual location of the defect, leaving the actual *debugging* (finding and correcting the defect) to be done essentially manually, with the aid of tools that help examine program execution and detect common errors [23]. Exhaustive model checking is often applied to concurrent programs, but this requires that the entire system be modelled within the verifier and large amounts of memory and processing time.

Programming assignments are assessed automatically in systems such as BOSS [22] or CourseMarker (formerly CourseMaster) [13] by executing tests on the code to be assessed and assigning a grade based on the number of tests that passed [1]. With larger programming assignments, this technique is usually used as a supplement to manual assessment instead of a substitute [1]. Testing does not work as well with concurrent programs, as the relative timing of the execution of different operations can have a critical effect on both the desired and the actual behaviour of the program. For this reason, manual assessment (using tests and/or model checking to check functionality) or requiring students to apply model checking to their own designs before implementation (e.g. [5]) seems to be favoured for assessing "real-life" concurrent programs. However, automatic assessment has been used for small and clearly delimited concurrent programming assignments such as solving the reader-writer problem or the producer-consumer problem in the SYPROS [12] intelligent tutoring system. SYPROS goes beyond mere assessment to providing detailed feedback tailored to each student while he tries to solve the assignment.

One of the problems with automatic assessment is that it is hard to design tests that detect all common errors and distinguish between different types of error without empirical data on real students' errors. This research into error types and frequencies in concurrent programming assignments is intended to mitigate this problem.

1.2.4 Testing and Debugging Tools and Methods

Current debuggers do not appear to fully make use of potentially useful visualisation and interaction techniques; most have very limited visualisations and many provide only a graphical replacement for the traditional textual user interface. A lot of visualisation research has been done that involves exploring new visualisation techniques based on what the researchers feel would be useful or filling a niche in a taxonomy rather than studies of the requirements of programmers (see e.g. [14]). Most debuggers can

only show the current state of the program, even though the cause of a program malfunction usually lies in the past. Concurrency also makes debugging harder, as concurrent processes often interact in unexpected ways. These problems combine to make it hard, even with a debugger, to find bugs. Only a few debuggers (e.g. RetroVue [7]) are specifically designed to aid in debugging concurrent programs, and they do not seem to be widespread.

Having quantitative data on programming errors would provide a background against which debugging methods and tools could be developed that address common real-world problems [21]. One foreseeable problem with using data from university programming assignments is that the data do not reflect the skills of professional programmers. This can be mitigated by using advanced university courses.

2. SETTING

This study is centred around the Concurrent Programming course at Helsinki University of Technology[1] in Autumns 2005 to 2008. The goal of this course is to teach students the principles of concurrent programming: synchronisation and communication mechanisms, concurrent and distributed algorithms and concurrent and distributed systems. Most students major in Computer Science or a related subject such as Mobile Computing and have completed a Bachelor's degree or a roughly equivalent part of a Master's degree.

This paper describes a study of the defects in the programs students wrote in the three mandatory programming assignments[2] of the Concurrent Programming course at Helsinki University of Technology during the autumns of 2005, 2007 and 2008. Due to differences in grading, the autumn 2006 instance of the course has been left out. All of the assignments were to be done in Java. Students could choose to do the assignments alone or in pairs; in both cases, the grading was the same. In 2005, students were allowed to retry the assignment several times. This was reduced to one resubmission in 2006. Resubmission was eliminated completely in 2008; the grading was made less severe and a test package provided to students to compensate.

Students were required to submit both the actual program source code and a brief report outlining how their solution works with an emphasis on concurrency.

As the students were required to submit their solutions through a WWW form that compiled their code, all the submissions were valid Java programs. The last submission by a student or pair of students before the deadline was assessed. Only submissions done before the initial deadline have been examined in this research; late submissions and resubmissions after receiving a failing grade have been left out.

2.1 Trains

In the first assignment (*Trains*), the students are given a simulated train track with two trains and two stations. The students' task is to write code that drives these trains from one station to another by receiving sensor events and setting the speed of the trains and the direction of the switches on the track.

2.2 Reactor

The second assignment (*Reactor*) is about the Reactor design pattern [24]. The students' task is to, using the synchronisation primitives built into the Java language, implement a dispatcher and demultiplexer that can read several handles that have blocking read operations at the same time and sequentially dispatch the events read from these handles to event handlers. The students then implement a simple networked Hangman game that uses this Reactor pattern implementation.

2.3 Tuple Space

In the third assignment (*Tuple space*), the student implements a simple tuple space [9] containing only blocking `get` and `put` operations on tuples implemented as `String` arrays. They are to do this using Java synchronisation primitives and use this tuple space implementation to construct the message passing part of a distributed chat server.

3. METHODOLOGY

The process applied here consists of three separate phases: data collection, defect detection and defect classification. They are described in this section.

3.1 Data Collection

The obvious source of information on defects in students' programs is the programs themselves. Furthermore, since students' programming assignments are graded by checking them for defects, the grading process already incorporates much of the necessary defect detection work.

Initial experiments with Java PathFinder [28] in which the model checker failed to complete verification even of simplified versions of the programming assignments described here, encouraged the use of testing to support our grading. Hence, the choice was made to assess the programs manually, essentially by reading the code and the students' explanations of it and checking whether it is correct. Testing was used to find situations that the programs did not behave correctly in.

This work was done primarily by hand by myself and assistants working according to specifications I provided and whose work I checked and, as needed, assisted with. This classification is explained further in Subsection 3.2.

3.2 Defect Classification

In order to serve the requirements of both teaching and tool development, I have classified the defects found in the students' programs using two separate classifications. One classification is by the underlying error (to the extent it can be determined), which helps determine what understanding or skill is lacking in the student who introduced the defect. In the other classification, defects are divided based on whether the program failures they cause occur deterministically.

Note that apparently non-functional requirements (such as using a mechanism that is not available) can be classified in this way by considering the execution of a call to a forbidden feature as a failure or by considering the operation to behave incorrectly. Since such requirements are typically based on a notional execution environment, it is natural to use the failure induced by this type of error in such an environment for classification purposes. This also makes this classification by failure consistent when the limitations of a notional environment are artificially introduced in the real environment, as in our Concurrent Programming course.

Defects and failures are defined here with respect to the written assignment specification, as interpreted by the person assessing the assignment.

[1] Since 1 January 2010, this is the Aalto University School of Science and Technology.

[2] For details, see the course web sites at: `http://www.cs.hut.fi/~jlonnber/T-106.5600.html`

3.2.1 Classifying Defects by Error

Errors can be classified by the task the programmer was performing when he made the error. This allows one to easily determine the knowledge and skills involved and provide feedback to the student to help him or her understand his or her error.

Inadequate testing can be considered a separate problem as it does not introduce defects into the code, although it (by definition) may prevent defects from being found.

I initially formed this classification by grouping together defects based on similarities in how they deviate from the corresponding correct solution; this is conceptually similar to the goal/plan analysis of Spohrer and Soloway [26]. However, instead of constructing a full goal/plan tree for each program (which was found to be very time-consuming due to the size of the programs involved and not very useful), only the incorrect parts of the program are considered. Defects are classified by the incorrect or missing subgoal or subplan in the most specific correct goal or plan (assuming top-down development, this means the students' errors are assumed to be made as late as is plausible in the development process). Most defects can be explained this way as errors in a specific plan or goal. Similarly, goals are considered equivalent if a plan that achieves them both is known. Plans are differentiated by their subgoals. While this greatly decreases the amount of different errors, this occasionally results in two otherwise correct plans interfering with each other; these errors are handled separately, as are cases where the students' plans cannot be determined. With some minor refinements and additional defect classes, this classification was used as a basis for assessment in 2007 and 2008.

Previously, I performed the analysis of defects [17] using only the students' programs and reports as data and constructed a classification schema based on the assessment criteria of the Concurrent Programming course at the time and on defect classifications found in the literature, especially the classification of Eisenstadt [8]. The top level of classification in that analysis was a division into:

Concurrency errors Misconceptions or design errors related to concurrency

General programming errors Misconceptions or errors related to the programming language or non-concurrent algorithms

Environment errors Errors related to the environment in which the assignment was performed

Goal misunderstandings Misunderstandings of the requirements of the assignment

Slips Slips or other careless errors

One problem with this classification was that only a small amount of the students' errors could be unambiguously placed in one of the above categories; only 23 %, 45 % and 34 % for the respective assignments. This was because asking students to explain the reasoning behind their entire solution in a written report did not give enough information to reconstruct their errors. Another reason was that some errors can fit into many classes.

Because of this, a phenomenographic analysis was done [19, 20] to provide an understanding of how students understand concurrent programming in order to analyse their defects meaningfully. The resulting phenomenographic outcome spaces led to some changes to the classification. While it would be possible to distinguish between errors made in designing the solution and implementing it, students did not consistently make this distinction [20, Table 3]. For this reason, it is hard in some cases and not very useful to make this distinction. The distinction between concurrency and general programming errors is similarly ignored. One reason is that, in a concurrent programming assignment, most programming errors are in some way related to concurrency; the question of where to draw the line has no clear answer. Another reason is that the phenomenographic study did not show that students make this distinction. Some did, however, show an awareness of the difference between deterministic and nondeterministic errors [20, Table 4]. Understanding the requirements of the assignment can be seen as a source of difficulties that is great enough to structure one's work around [20, Table 3]. Alternative understandings of the goal of an assignment, which lead to understanding the requirements differently, exist [20, Tables 1 and 2].

The distinction between the programming and the assignment environments is made in order to determine which errors are irrelevant in assessing the students' concurrent programming knowledge and skill and could be reduced or eliminated by changing the assignment.

Requirement-related error A programmer can fail to understand part of a specification correctly or fail to take it into account properly when designing or implementing his solution. Some understandings of the goals of a programming task (e.g. seeing a passing grade as the goal of a programming assignment) can lead to this. Pointing out the requirement and a failure in which it is violated should be enough to explain this type of error to the programmer. Communicating requirements as tests with a clear pass/fail indication can help programmers detect these. Eliminating this type of error should be a priority when designing programming assignments.

Programming environment-related error Some misconceptions of the goals of a programming task that relate to the target environment, such as considering unbounded memory usage to not be a problem, can result in this type of errors. Alternatively, there may be something about the language, API or other aspects of the execution environment the programmer has not understood, in which case explaining the relevant aspect (e.g. by referencing a specification) may help. Finding problems in students' knowledge of a programming environment in general can be helpful to them, but secondary in many advanced courses to the actual topic of the course, such as concurrent programming.

Assignment environment-related error Misconceptions about the framework provided for a programming assignment can also result in errors. These are distinguished from errors in the previous category in that they relate to systems that are only used in this particular programming assignment. Therefore, these errors, like the requirement-related errors above, can be seen as indications of the assignment being confusing instead of lack of any understanding or skill relevant to concurrent programming in general. This type of error is avoided if no framework is provided (as in the Reactor assignment); large amounts of this error suggest that the framework is confusing and should be simplified.

Incorrect algorithm or implementation Programmers may introduce errors when creating or implementing

an algorithm. These errors vary from creating an algorithm that does not work in all necessary cases to forgetting to handle a case. Showing a programmer how his code fails is enough if the error is not due to insufficient or incorrect knowledge. Since some students do not make a clear distinction between creating an algorithm and an implementation, these errors are grouped together. A programming assignment should allow students to make errors of this type, as they provide valuable indications of deficiencies in the students' knowledge or skill.

In each assignment, different subtypes of the aforementioned errors can be distinguished. They are described in the following to the extent they merit interest either by being common, surprising or illustrative of students' understandings of concurrent programming.

3.2.2 Classifying Defects by Failure

An alternative classification is by the type of failure; this is relevant for testing and debugging. Some students showed an awareness of this distinction [20, Table 4].

Deterministic failures occur consistently for a given input (or sequence of stimuli in the case of a reactive program) and are thus easy to reproduce. This allows traditional debugging, based on repeated executions, single-stepping and breakpoints and examining program states, to be used. History-based debugging methods can also be used.

Nondeterministic failures are hard to duplicate; a logging-based debugging approach is therefore more useful than traditional debugging, since the failure only needs to occur once while logging is being done.

Since the debugging technique must be chosen based on the symptoms and nondeterministic failures may appear to be deterministic in many tests, it makes sense to always use techniques appropriate for nondeterministic failure when debugging concurrent programs.

This classification was done by examining the effect of each defect class on program execution through testing and reasoning.

4. RESULTS

In this section, I describe the defects found in the students' programs in terms of the defect classifications described in Subsection 3.2. Detailed lists of defects are also available [18].

4.1 Trains

An interesting aspect of the Trains assignment (described in Subsection 2.1) is that, since the train simulation combined with the student's train control code takes no input from outside, almost all failures are nondeterministic; a deterministic failure would occur in every possible execution, making it easy to detect. It is therefore not surprising that all the deterministic failures are due to misunderstandings of the requirements. Since the concurrent programming aspect of the assignment is easy in comparison to the other assignments, it is hardly surprising that most of the students' errors are related to the simulator and what they are supposed to do with it. Figure 1 shows, for the three yearly instances of the course that I have analysed, the total amount of submitted programs and the amount of defects found in each class in both the error- and failure-based classifications.

4.1.1 Requirement-related Errors

The train simulator used in the Trains assignment proved to have some confusing aspects in its original form used in 2005. Particularly problematic was that the students' code could easily access information about the simulated trains that was not supposed to be available. The trains could also communicate with each other in ways that students were not allowed to use in the assignment, such as shared variables. This allowed students to avoid much of the expected semaphore usage. The assignment also required students to implement the required random delay at stations themselves, which in many cases was replaced by a fixed delay. These problems were eliminated in the 2006 version of the assignment by redesigning the simulator and its API so that the options available to the student in the simulation environment matched the requirements.

After this, the most common form of requirement-related error (accounting for almost all of the requirement-related errors in 2008) is that at least one train uses the secondary choice for a track or station platform even when the primary choice is free, ignoring the requirement to use the upper platform or shorter track where possible. This requirement exists to prevent statically allocating one alternative to each train, removing the need for choosing between alternative tracks. However, it is vague, hard to test for (our test package does not detect it) and overlooked by a few students every year.

4.1.2 Programming Environment-related Errors

In three cases in 2005, students had clear misunderstandings of the Java language or API, such as accidentally generating negative random numbers or leaving out the `break` statement at the end of a `case` of a `switch` and insisting that the fall-through is a compiler bug. Before 2004, introductory programming was taught at Helsinki University of Technology using Scheme instead of Java, so some students may have been unfamiliar with Java.

4.1.3 Assignment Environment-related Errors

The train simulator used in the first assignment proved to pose problems of its own by introducing issues of train length, speed and timing that cause problems for students unrelated to the learning goals of the assignment and hence distract the student from the concurrent programming the assignment is about. Some of the rules of the simulation were also not obvious to the students.

By far the most common type of error here was placing the sensors used to release a track segment too near a switch, allowing the other train to enter or change the switch before the first has left. This type of error has decreased since 2005, probably because the simulator and its documentation have been revised for clarity several times. Other sensor-related issues, ignoring the crossroads at the top of the track and setting the trains' speed too low account for the rest of the errors in this category.

4.1.4 Incorrect Algorithm or Implementation

Almost all the solutions were close enough to being correct for specific problems to be identifiable. Most of the errors were found in the train segment reservation code. Some solutions consisted of subsolutions that did not combine properly or relied on train events happening in a specific order; there is no indication that the students considered the possibility that the order could be different. Others had more localised problems, such as changing switches at the wrong time or not at all, using the wrong semaphore or the right semaphore at the wrong time or

	2005	2007	2008
Submissions	128	60	52
Requirement	53	10	11
Programming	3	0	0
Assignment	70	20	10
Incorrect	28	16	5
Deterministic	39	2	0
Nondeterministic	115	44	26
Total	154	46	26

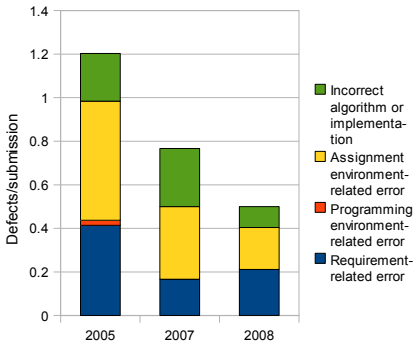


Figure 1: Defects found in Trains assignment

initialising a semaphore to the wrong value; many of these appear to be implementation slips since a correct solution is described and similar situations are handled correctly in the same program. A few unnecessarily complex solutions introduced the possibility of deadlock by ignoring the possibility of a sequence of semaphore operations being interleaved with operations made by the other train.

Only a few errors were obvious implementation slips, such as forgetting a `break` or `else`, matching sensors incorrectly, parenthesising a logical expression wrong, making an array one element too small or accidentally duplicating or commenting out code.

4.2 Reactor

The Reactor design pattern in the second assignment (see Subsection 2.2) turned out to be hard to understand for some students; in many cases, the students' programs are correct solutions to what they consider to be the problem. Clarifying the intent and structure of the Reactor pattern was clearly necessary, so I wrote a simplified explanation of the Reactor pattern for the next year's course.

The defects found are summarised in Figure 2. The increase in defects between 2005 and 2007 can be mostly ascribed to more aspects of the programs being taken into account in assessment, such as memory use.

4.2.1 Requirement-related Errors

Until students were provided with a test package in 2008, many made changes to the Reactor API or the way it uses threads to simplify the Reactor or the Hangman server. These errors account for roughly a third of the requirement-related errors. Similarly, problems with input and output formats and the rules of the Hangman game were common until the test package was introduced.

The most commonly ignored requirement was to ensure that the Reactor does not buffer an arbitrary amount of data if it cannot handle events quickly enough. In 2005 and 2006, this was not considered a problem, but in 2007 and 2008 it was found to occur in the majority of submitted solutions. This error by itself accounts for more than three quarters of the requirements issues found in 2008. The fact that it remained common in 2008 is probably due to the fact that the test package did not include a test case for this scenario.

A few of the submitted Reactor implementations in 2005 submitted all events to all event handlers. It was found that Schmidt's pseudo-code for the Reactor implementation [24] can also be interpreted this way; for the 2006 course, I wrote a simpler explanation of the Reactor pattern that eliminated this ambiguity. A similar ambiguity involved the amount of events to dispatch for each call to `handleEvents()`. Using busy waiting or polling in the Reactor or Hangman and failure to terminate properly accounts for the remaining cases.

The sharp decrease in deterministic errors in 2008 is almost entirely due to failures to comply with the specified APIs and I/O formats (about 80 % of the deterministic errors) being essentially eliminated by the test package.

4.2.2 Programming Environment-related Errors

In 2005, the console I/O required by the Hangman client was by far the most problematic aspect of the programming environment. The client was deemed unnecessary and removed the next year. Several cases of using a fixed TCP port number when required to use a free one as shown in the example code have been found.

Four cases in 2005 were due to misconceptions about Java.

4.2.3 Incorrect Algorithm or Implementation

Many solutions, especially in 2007, failed to correctly handle events that were left undispatched after handle removal or received after handle removal; again, there is no indication that these students considered this sequence of events. Some failed in other ways to correctly remove handles from use. The increase in 2007 may be, like the previous error, due to improved assessment guidelines. Again, the testing package makes this type of error easier to detect.

Several different cases were found of incorrect buffer management algorithms in the Reactor implementation. In some cases, status variables were set at the wrong time or not at all. Two circular locking dependencies were found, which can be seen as two subsolutions conflicting. In some solutions, the defect involved notifying the wrong thread or at the wrong time; again, with no indication that such a possibility had been taken into account. In some case, messages were overwritten or lost, either due to possible interleavings not being considered or because the student did not consider it relevant to handle certain situations, such as messages appearing faster than they can be dispatched or before the main loop is entered. Only a few cases of using collections or variables without the necessary synchronisation were, however, found, mostly involving a flag variable or the list of active handlers not being protected by a `synchronized` block with no explanation given.

A few obvious implementation slips were found, such as having the Hangman server and client connected to different ports, starting the same thread twice, declaring

	2005	2007	2008
Submissions	107	51	40
Requirement	93	112	38
Programming	15	11	1
Incorrect	51	56	17
Deterministic	94	102	8
Nondeterministic	65	77	49
Total	159	179	57

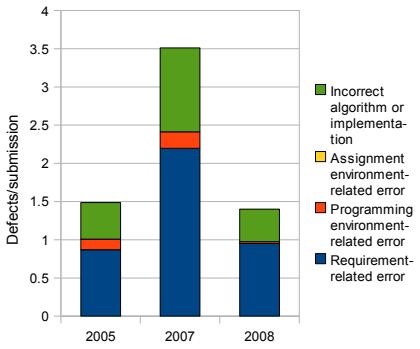


Figure 2: Defects found in Reactor assignment

an array that was one element too small and using a stack instead of a queue.

4.3 Tuple Space

In the tuple space assignment (described in Subsection 2.3), the requirements of the assignment were once again problematic in the 2005 original. However, many of the defects found were clear indications of careless or unskilled concurrent programming. By this time, the Java programming environment was apparently familiar to the students, as no clear misunderstandings of the programming environment were found. The defects found are summarised in Figure 3.

4.3.1 Requirement-related Errors

As in the first assignment, about half of the requirement-related errors in 2005 were due to the requirement to pretend that the chat system was running in a distributed environment. Making the corresponding error in later years and causing failures in the distributed context was much less common. There were fewer problems with the division between tuple space and chat system than between Reactor and Hangman. Polling occurred in a few cases in either the chat system or tuple space.

The most commonly ignored requirement of the chat system's functionality was that messages stay in order. As an example of the variety of other errors of this type, a few students in 2005 and 2007 allowed their chat system to combine messages stored in the log for delivery to new listeners into one message that looked the same to the user of the provided GUI, arguing that they could ignore the specification as long as the user experience is the same. Yet again, the test package seems to help students understand they have a problem.

The semantics of the tuple space also caused problems. Most of these errors involved limiting the tuples in some way, such as considering the first element in a tuple to be a `String` used as a key as in the textbook. Some solutions changed the blocking, matching or copying semantics of the `get` operation. One error of note of this type (which the test package did not detect) is storing references to tuples in the tuple space rather than copying their contents, which only 2 students in 2007 did, but 10 in 2008. This suggests that students rely on the test package to detect errors in conforming to requirements such as these.

Again, most of the decrease in deterministic errors in 2008 can be attributed to the test package helping students understand they have misinterpreted the specification (more than two thirds of the deterministic errors in 2005 and 2007).

4.3.2 Assignment Environment-related Errors

The GUI provided to the students to make the requirements easier to understand sends messages when listeners leave (and, in 2005, when they join) a channel; this caused some students to require this behaviour for their implementation to work.

4.3.3 Incorrect Algorithm or Implementation

The tuple space proved to be unproblematic to implement. Only a few cases of critical sections having the wrong extent and `notify()` being used instead of `notifyAll()` were found. More common was for the tuple space to match patterns against tuples incorrectly. A few solutions also corrupted their own data structures while executing, for various reasons including implementation slips, understanding returning an object to mean returning its contents and using library classes incorrectly.

Cleaning up after a handle is removed for use appears to often have problems, as does ensuring memory use stays within reasonable limits. Similarly, getting rid of unused tuples is a difficult area, accounting for roughly a third of the errors in this category. In some cases (especially those where no cleaning up is done at all), this could be because cleanup is not considered by the student to be relevant to the assignment (i.e. the intended execution environment is not understood to have limited memory). However, most of the reports of students with this error suggest an awareness of memory limitations and a choice to use a simple algorithm that wastes memory rather than a complex one that conserves it, suggesting this is a compromise to save time and/or decrease chances of a programming error.

Initialisation proved to be surprisingly problematic, especially, interestingly enough, the `ChatServer` constructor for connecting to an existing chat system, which often did not replace all the tuples it got. This invariably causes the system to grind to a halt when the third server node is connected. Outside this method, forgetting to replace tuples was uncommon.

The buffer of messages that the chat system has to maintain for each channel proved to be problematic, with failure to handle a full buffer or simultaneous writes, insufficient locking of the buffer or related sequence numbers and indices being common in 2005 and 2007. The test package may account for the decrease in 2008. Circular locking dependencies, on the other hand, became much more common in 2008, typically in the form of the locking for different operations, such as writing messages and closing listeners, interfering with each other.

5. DISCUSSION

	2005	2007	2008
Submissions	84	49	39
Requirement	93	49	21
Assignment	3	0	0
Incorrect	70	51	36
Deterministic	98	58	28
Nondeterministic	68	42	29
Total	166	100	57

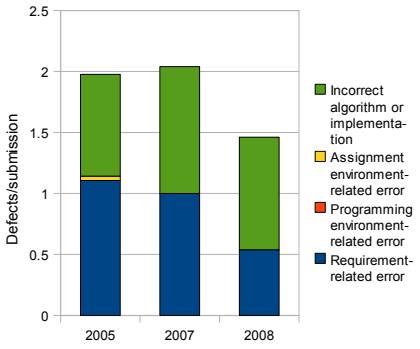


Figure 3: Defects found in Tuple space assignment

This study suggests several areas in which students have problematic understandings that lead to incorrect concurrent programs. These problematic understandings are related both to assignment goals and to the concurrent programming concepts or development practices that are being taught.

The quantitative results appear to show dramatic decreases in certain types of defect in the students' programs as the intended result of certain changes to the assignments. In particular, providing tests helps students notice their problems with understanding assignment requirements. It is possible that other changes to the course (for example, changes to the resubmission policy) or the participating students (for example, the course no longer being mandatory except for international master's students) may also have affected the results.

5.1 Understanding Program Execution

The large amount of defects found in students' programs that cause failures nondeterministically is not surprising, since these defects are both hard to find and correct. Testing software that helps make these defects manifest will help students find such defects by themselves. The results of the Reactor assignment seem to bear this out. However, no such dramatic improvement can be seen in the Tuple space assignment. One plausible reason for this is that the students were not capable of debugging their programs despite knowing that they contain bugs. Reasons cited by students include not understanding the tests and being unsure whether the tests timed out due to deadlocks or their code being too slow. The latter can be mitigated by providing debugging tools that can clearly show pending and previous operations on semaphores, monitors and tuple spaces, allowing students to determine what the exact failure is.

Many students introduce defects in their programs that appear to be caused by misunderstanding or reasoning incorrectly about concurrent program execution. In particular, many difficult concurrency bugs the students introduce appear to stem from two different parts of their programs interacting badly. Students should either be encouraged to consider their program as a whole or design it in such a way that interaction between parts is minimised. Another common source of bugs is that some possible orderings of events have not been taken into account. It may be helpful to increase the emphasis on designing programs to avoid unexpected interactions between processes. In both cases, the bugs can also be found, naturally, during verification.

Part of the problem is that the runtime behaviour of a concurrent program, a necessary part of the programmer's perspective, is hard to examine or interpret, preventing students from effectively understanding what their program does and reasoning in terms of the relevant concurrency model. Another possible problem is that the models of concurrency used in textbooks such as the one by Ben-Ari [4] used in the course do not match the concurrency model of e.g. Java [10] in all the relevant aspects. For example, Java allows compilers and multiprocessor architectures to reorder operations within a thread as long as all the operations *within this thread* produce the same result. This means that other threads may read combinations of values of variables that are impossible in textbook concurrency models. To address this, I suggest a greater emphasis in teaching concurrent programming on real-world concurrency models than the aforementioned textbook models. In order to understand how their programs fail, students should be shown how their programs really behave so that they can realise that their understanding of concurrency is incomplete and correct it.

I suggest that what students need to effectively understand what their concurrent programs do is a tool to generate execution history visualisations automatically from a running program that are easy to understand and navigate and provide the information needed by the student in an easily understandable form. The large amount of nondeterministically manifesting defects in students' programs demonstrates a clear need for debugging tools that do not rely on repeated execution and stepping as is the traditional approach. Instead, the information needed for debugging should be captured for post-mortem examination from a failing execution when it occurs.

Giving students tools to study memory allocation would help them understand how their programs use (or misuse) memory. In its most basic form, this could involve using a profiler to get information on the maximum memory use of their program. More detailed visualisations, such as charts that show memory use over time categorised by where the memory is allocated, can be used to help students understand memory use in more detail. Other resource usage issues, such as use of CPU time or network or disk capacity, can be addressed using similar visualisations.

5.2 Verification

Students have a wide range of approaches to testing. Some students used completely unplanned, cursory, testing. Some tried to 'break' the system, while others covered a variety of different cases. Moreover, some students found they cannot test their program adequately by themselves

and need help from another person or tool, that testing in itself is not sufficient or that you have to prove your program correct by hand. [20, Table 4]

The students' verification approaches could be improved by providing testing tools to generate scenarios that are hard to discover using normal testing procedures and more explicit and detailed guidance on how to apply different verification techniques in practice. The assignment itself could be changed to encourage students to learn and apply different verification techniques by explicitly requiring models, as done by Brabrand [5], or by requiring students to create suitable tests, e.g. using test-driven development.

Adapting programs to a model that can be checked using a model checker is often hard and error-prone work. This makes this approach especially impractical for students to use in an assignment unless the modelling of their solution is in itself a goal of the assignment as in Brabrand's course above or the assignment is carefully designed to facilitate efficient model checking.

An alternative approach to finding concurrency bugs is to increase the chance of interleavings that lead to failure. Stress testing is a well-known approach, and its usefulness can be further improved by making sure interleavings occur often and in many places. One straightforward and realistic approach is to distribute the program's threads over multiple processors. Another way to do this is to automatically and randomly change the thread scheduling to make concurrency failures more likely to occur (e.g. [27]). This is the approach used by the automated testing system of our concurrent programming course.

5.3 Communicating Goals

Students may have a different understanding of what they are trying to achieve than their teachers. Many of the students in this study wrote programs that were missing required functionality or implemented this functionality in ways that conflicted with requirements or required additional limitations on the runtime environment. One reason we found for this was that students had different aims in their assignment, seeing it primarily as something they have to do to get a grade or as an ideal problem in an ideal context in which simplifying assumptions apply [20, Table 1]. The students also considered different potential sources of problems: the hypothetical user of the program (even when the assignment was specified in terms of the input and output of methods, not user requirements), underlying systems that could fail, especially network connections in a distributed system, and the programmer (the student) as a error-prone human [20, Table 2].

These purposes of the programming task and sources of failure of the students suggest that many of the errors made by students are misunderstandings of what their program is supposed to do and what situations it is expected to cope with rather than actual misunderstandings of concurrent programming itself. It is hard for a student to discover such problems by himself if all he has to go on is a specification in natural language that is open for several different forms of misinterpretation.

In a course that, like our Concurrent Programming course, has as its goal to teach students software implementation techniques with an emphasis on reliability and correctness, it is desirable to have programming assignments with clear and specific goals. One reason is to guide the students into applying the techniques that they are expected in the course to learn to apply. Another reason is that it is hard to say how correct a program is if it is not clear what it is supposed to do. Hence, requirements should be self-contained in the sense that they should be unambiguous and not require specific knowledge of a (hypothetical) usage context or users. The teachers and students can then focus on issues more relevant to the learning goals of the course, such as correctness and efficiency. Finally, if the requirements are specific enough to be expressed as test cases or some other form that can be checked automatically, it is much easier to use automation to assist in assessment and in helping students determine whether they are solving the right problem and whether they are solving it correctly. All this suggests that teachers should, when designing programming assignments for implementation-oriented courses, make assignment goals more explicit and concrete. Naturally, in courses that are intended to teach students to determine user requirements or design systems to meet user requirements, this approach is not applicable; there is a clear need for students to be able to cope with vague or unknown requirements.

One important aspect is that the goals should specify *what* the student should achieve rather than *how*, allowing students to find their own solutions to problems. The student should be able to see his program clearly fail to work correctly rather than be told afterwards that he did something the wrong way or failed to take a usage scenario into account.

Two different types of measures have been taken on our Concurrent Programming course to address these issues. One was to change the environments in which several of the assignments were done to make limitations more concrete, such as actually making the Trains and Tuple space assignments function as distributed systems (in the form of separate processes) rather than as threads within one virtual machine. The other major change was made after several students each year requested that they be given access to the package of tests for the assignments used by the course staff to support assessment. Giving the students a test package that clearly states whether their solution fulfils the specification's demands appears to have decreased the amount of errors even in assignments where students had easy acccess to tests, such as Trains. Naturally, giving students pre-written tests can easily eliminate their motivation for designing their own test cases. Introducing a test package is similar to introducing automatic assessment and allowing students to resubmit each assignment many times. Even when unlimited access to automatic assessment has been given, it seem that only a small minority of students make use of repeated reassessment rather than trying to correct their mistakes independently [2].

6. CONCLUSIONS

The analysis of defects found in students' concurrent programs described in this paper shows that students often have difficulties understanding requirements and taking them into account and in noticing defects that lead to nondeterministic failures. It seems that both issues can be addressed by providing students with test packages that show them how their programs fails to meet requirements. Nondeterministic execution is also difficult for students and debugging tools based on capturing and visualising execution histories can help address this.

7. ACKNOWLEDGEMENTS

I'd like to thank the teaching assistants who did much of the hard work of finding the students' bugs: Teemu Kiviniemi, Kari Kähkönen, Sampo Niskanen, Pranav Sharma, Yang Lu, Ari Sundholm and Pasi Lahti. I'd also like to thank the people who've given me feedback on this work, in particular Lauri Malmi and the reviewers.

References

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] K. Ala-Mutka and H.-M. Järvinen. Assessment process for programming assignments. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, pages 181–185, 30 Aug.-1 Sept. 2004. doi: 10.1109/ICALT.2004.1357399.

[3] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2 edition, 1990. ISBN 1850328803.

[4] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education, second edition, 2006.

[5] C. Brabrand. Constructive alignment for teaching model-based design for concurrency. In *Proc. 2nd Workshop on Teaching Concurrency (TeaConc '07)*, Siedlce, Poland, June 2007.

[6] I. Burnstein. *Practical Software Testing*. Springer, 2003.

[7] J. Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[8] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/248448.248456.

[9] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[11] L. Grandell, M. Peltomäki, and T. Salakoski. High school programming — a beyond-syntax analysis of novice programmers' difficulties. In *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, pages 17–24, 2005.

[12] C. Herzog. From elementary knowledge schemes towards heuristic expertise — designing an ITS in the field of parallel programming. In C. Frasson, G. Gauthier, and G. I. McCalla, editors, *Proceedings of 2nd International Conference on Intelligent Tutoring Systems*, volume 608 of *LNCS*, pages 183–190. Springer, June 1992.

[13] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, pages 46–50. ACM Press, 2002. ISBN 1-58113-499-1. doi: http://doi.acm.org/10.1145/544414.544431.

[14] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.

[15] IEEE. IEEE standard classification for software anomalies. Technical Report Std 1044-1993, IEEE, 1994.

[16] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[17] J. Lönnberg. Student errors in concurrent programming assignments. In A. Berglund and M. Wiggberg, editors, *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling 2006*, pages 145–146, Uppsala, Sweden, 2007. Uppsala University.

[18] J. Lönnberg. *Understanding students' errors in concurrent programming*. Licentiate's thesis, Helsinki University of Technology, 2009.

[19] J. Lönnberg and A. Berglund. Students' understandings of concurrent programming. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 77–86. Australian Computer Society, 2008.

[20] J. Lönnberg, A. Berglund, and L. Malmi. How students develop concurrent programs. In M. Hamilton and T. Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, pages 129–138. Australian Computer Society, 2009.

[21] J. Lönnberg, L. Malmi, and A. Berglund. Helping students debug concurrent programs. In A. Pears and L. Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*, pages 76–79. Uppsala University, 2009.

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

[23] R. C. Metzger. *Debugging by Thinking*. Elsevier, 2004.

[24] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[25] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student errors in algorithm simulation exercises. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 81–86. University of Joensuu, November 2005.

[26] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6138.6145.

[27] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.

A Note on Code-Explaining Examination Questions

Simon
University of Newcastle
PO Box 128, Ourimbah
NSW 2259 Australia
+61 2 4348 4074
simon@newcastle.edu.au

ABSTRACT
The BRACElet project, which explores aspects of the way students learn to program, involves questions to assess the students' skills in tracing, reading, and writing code. In a recent examination based on the BRACElet specification, students' marks were significantly lower on the code-reading questions than on the other two types. A close examination of the students' answers leads to the conclusion that a marking scheme for code-reading questions should be proposed explicitly, and that work is still required to ensure that students fully understand what sort of answer will earn them full marks for such a question.

Categories and Subject Descriptors
K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education.*

General Terms
Measurement, Experimentation, Human Factors.

Keywords
Novice programmers, CS1, tracing, comprehension, SOLO taxonomy.

1. INTRODUCTION
The McCracken working group of ITiCSE 2001 [7] provided evidence that students' inability to program after a first programming course was, if not universal, at least widespread and apparently independent of the nature of the institution at which the students were studying.

The Leeds working group of ITiCSE 2004 followed this up with an investigation into student programmers' code-reading skills. "Even when our principal aim is to teach students to write code, we require students to learn by reading code. In our classrooms we typically place code before students, to illustrate general principles. In so doing, we assume our students can read and understand those examples. When we exhort students to read the textbook, we assume that the students will be able to understand the examples in that book" [3]. In other words, it seems unlikely that we can claim to be teaching students to write code if they can't read and understand it.

Koli Calling '09, October 29 – November 1, 2009, Koli, Finland.

The BRACElet project [12] has for some years been exploring this relationship between student programmers' skills at reading code and at writing it. In recent papers [4, 6, 11] the question has been probed more deeply, with attempts to normalise the assessment of code tracing, reading, and writing skills, and to establish whether there is a recognisable hierarchy in the acquisition of these skills.

1.1 The BRACElet 2009.1 (Wellington) specification
While the BRACElet project was initially restricted to recognised project members, a new initiative in early 2009 saw the study design published [13], with an invitation to non-members to run their own studies and publish the results independently of the main project.

According to the study design, the code-reading and code-writing questions are to be accompanied by a set of basic skills questions, which typically take the form of code-tracing or desk-checking questions. These questions should establish whether students can show a basic knowledge of the programming constructs to be used in the reading and writing questions. If they cannot, it might be reasonable to expect that understanding code and writing code will be beyond them.

The BRACElet 2009.1 (Wellington) specification [13] requires that students be tested on the following:

- Tracing questions, in which students are expected to trace through code with specified values. There are to be three types of tracing question: one non-iterative and non-recursive, one iterative without control logic within the loop, and one iterative with control logic within the loop.
- Reading/understanding questions, in which students are required, for example, to determine the purpose of a piece of code. These questions must not use the same code as the tracing questions, or students might use the tracing output to determine the purpose. They should also have some uninitialised variables (such as parameters), to make tracing harder. At least one question must be similar in complexity to one of the writing questions.
- Writing questions, in which students are to write a piece of code that satisfies given specifications. At least one of these questions should be of similar complexity to one of the tracing questions, and at least one should be of similar complexity to one of the reading/understanding questions.

A BRACElet paper from the same conference as the Wellington specification explored a possible finding that students might actually be learning to write code before they could read it [9]. Unable to find statistical evidence for or against the finding, the paper left the question open, but it also made in passing the tantalising observation that "it would (probably!) not be reasonable to ask students in one question to read and understand a given piece of code, and in another to write the same piece of code."

The exam reported on in this paper has a set of questions based on the BRACElet 2009.1 (Wellington) specification [13], but also rises to the challenge of the foregoing observation. It includes the three tracing questions, as specified. It also includes three reading/understanding questions meeting the same specifications, and three writing questions meeting the same specifications. Thus there are three non-iterative non-recursive piece of code, virtually but not entirely identical, one in a tracing question, one in a reading question, and one in a writing question; there are three similarly related iterative questions without control logic within their loops; and three similarly related iterative questions with control logic within their loops. To experienced programmers the three questions in each set might be effectively identical, but perhaps they would not appear so to a novice student.

In setting the exam I expected that the better students would see the correspondence between the questions in each set, and would thus, for example, be able to answer the code-writing questions by modifying the code provided in the tracing and reading questions. But the better students are the ones who would least need to do that, so perhaps it really was possible, for all intents and purposes, to "ask students in one question to read and understand a given piece of code, and in another to write the same piece of code."

1.2 The role of SOLO in BRACElet

The SOLO taxonomy [1] was introduced to the BRACElet project in 2006 [5, 14]. SOLO categorises a student's answer to a question not according to its correctness but according to its level of abstraction. A student's answer can be classified as

- Prestructural: essentially showing no idea of what the question is about;
- Unistructural: showing some knowledge of one particular aspect of the question;
- Multistructural: showing understanding of all parts of the question, but not of how they synthesise into a whole;
- Relational: showing a holistic understanding of the question;
- Extended abstract: going beyond a holistic understanding of the question to place the answer in some hitherto unspecified context.

The essential finding of BRACElet papers using the SOLO taxonomy is that the more abstract a student's answers to code-reading questions, the more likely the student is to perform well on code-writing questions. When asked to explain the purpose of a piece of code, students who give relational answers tend to be better programmers than students who give multistructural answers. In this sense a relational answer to a code-reading question is a 'better' answer than a multistructural, unistructural, or prestructural one [4, 6].

2. THE COURSE AND THE EXAM

The introductory programming course from which this data is drawn runs for a single semester of 12 teaching weeks, and its final assessment item is a three-hour written exam to which students are not permitted to bring reference materials such as notes and books. The programming language used in the course is Visual Basic. The nine questions that were used for this analysis are presented in Appendix A.

One of BRACElet's early concerns about code-reading questions was that students might not understand what was being asked of them, that they might not understand what type of answer was being sought, either because the wording of the question was inherently ambiguous or because students had not had sufficient exposure to such questions.

With regard to the concern about ambiguity, rather than the early question wording of "In plain English, explain what the following code does", I used the words "Explain what the following code does. You are not being asked to explain each line of the code; you are being asked to explain its overall purpose."

To offset the concern about students' familiarity with the question type, the weekly exercises in this course always included a code-explaining question, generally with the same initial wording that was to be used on the exam questions.

While it was not possible to collect accurate data on this point, anecdotal evidence suggests that when doing the weekly exercises the students almost invariably skipped this question (which was always the first question in the set). When asked why, they explained that it was too hard to read and understand code; they would rather write it.

This offering of the course is at the university's campus in Singapore, where the official language is English but other languages are widely spoken. The course caters both to Singaporean students and to students from other countries where English is not the first language. Of the 57 students who consented to take part in the study, only 10 indicated that they speak English at home. All of the students are enrolled in a computing degree; 23% of them are female; and their ages range from 17 to 34, with a median of 20 and mean of 21.

3. ANSWERS TO THE CODE-READING QUESTIONS

It was immediately apparent after the marking that students' marks on the code-reading questions were not as high as their marks on the code-tracing or code-writing questions. This was borne out by scatter plots of the students' marks in the three sections. Figure 1 shows the students' percentage marks in each of the three sections, plotted against their overall marks for the exam. If the problem had become apparent during the marking, the marking scheme might have been adjusted to remedy the problem, at least in part; but these exams were marked by the lecturer at the overseas campus according to a rubric developed by the lecturer at the home campus, and the overseas lecturer followed the rubric faithfully.

While the scatter points of Figure 1 are difficult to interpret visually, the trend lines appear to support the observation that students score comparable marks on the code-tracing and code-

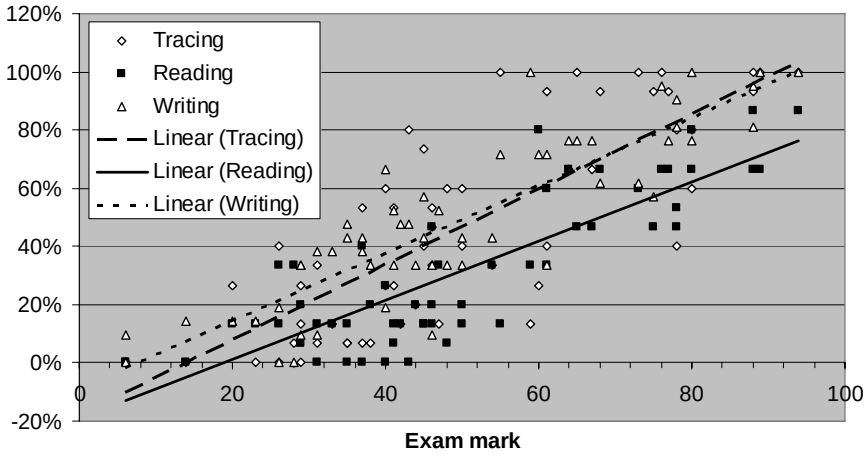


Figure 1: Students' marks in tracing, reading, and writing, plotted against overall exam mark

writing questions, and lower marks on the code-reading questions. Paired two-sample t-tests showed that there is no significant difference between the mean marks of the tracing questions (mean 46%) and the writing questions (mean 49%), but that both differ significantly from the mean mark of the reading questions (mean 31%), with $p<0.01$.

Why, then, do students earn lower marks for these questions? If they can show some understanding of code by tracing its execution, and can write code, why can they not determine and explain the purpose of a small code segment?

Any attempt to answer this question must be informed by a close examination of the code-reading questions themselves – which, as mentioned earlier, are provided in Appendix A. An underlying assumption to this approach is that the code-tracing and code-writing questions are not problematic. This is not guaranteed, but the assumption appears reasonable on the basis that I have used both of these question types for many years – they are known and familiar – while this is my first use in exams of code-explaining questions, so the lower marks make them a clear target for investigation.

3.1 Q26: assignment within If within loop

Question 26 asks the students to determine the purpose of a code segment consisting of an assignment statement within an If statement within a For loop. After initialisation the loop works through an array of strings called strTitle, assigning to a variable strOne any element of the array whose length is greater than that of strOne. When the loop has finished, strOne will have the value of the longest title in the array. The purpose of the code is thus to find the longest title in the array.

I believed this to be the most difficult of the three code-reading questions. This is partly because of its combination of structures, but also because in it I had unintentionally used two different inbuilt Length functions, one to determine the size of an array and one to determine the length of a string. Nevertheless, it was the one on which students scored best, although the average mark was only 1.72 / 5.

An ideal answer to the question is an accurate description of the code's purpose at a relational level in the SOLO hierarchy. Examples (preceded by the code numbers assigned to the students who wrote them) are

> **pd015:** After the given code statements are executed, strOne will have the element of the strTitle array whose length is the largest.
>
> **pd042:** This is a program to show the longest title available.

Some students came close, but with either a minor flaw…

> **pd051:** Store the longest lenght from the array strTitle to strOne
>
> (it is the title, not its length, that is stored)

or an unwarranted extension …

> **pd006:** The purpose is to find the longest length of string inside the title and locate it to the first place.
>
> (several students suggested this, perhaps misled by the deliberately uninformative variable name strOne)

Some answers were clearly accurate, but expressed in SOLO multistructural rather than relational terms:

> **pd020:** The program will compare each string that is store in the array with the variable strOne which contain the string in index 0 of the array.
>
> If the compare result shows that the length of the string in the index is greater than the string length store in strOne, then strOne will be overwrite by the longer length. The comparision will be done until the program has go through all the strings store in the array.

An answer of this sort was not awarded a good mark, as the express intent of the question was to determine whether students can take the next step beyond desk-checking code to determine what it achieves. I considered a SOLO relational answer to be a clear requirement of this and the other code-explaining questions.

However, a SOLO relational answer is not in itself sufficient, as illustrated by these incorrect relational answers:

> **pd025:** This code displays the titles with the most number of characters in descending order.
>
> **pd026:** It is to check which title of a string is longer than the previous title of a string.

pd031: To find out the total of the title.
pd033: It will display the value of strTitle in the reverse order.
pd034: The purpose of this code is to search for the required book title. If the index of strTitle is more than the required length, the book is found.

An incorrect relational answer such as those above can be caused by a slight error in understanding the code, a complete and utter guess at its purpose, or anything between. True misunderstanding is evident only when students attempt to explain their answers, either in addition to or instead of the expected relational answer. One student, for example, failed to recognise that parentheses indicate an array …

pd049: The overall purpose for the code is to compare the length between One and Title.
If the Title length is greater than One length, the strOne will be equal to the strTitle multiply with i.
According to the code, i is equal to 1 to strTitle.Length – 1.

Another student appears to think that the assignment symbol represents equivalence, akin to its meaning in mathematics:

pd056: When the value of i between 1 and strTitle.Length–1, the value of One and title are always the same while the length of title more than the length of One.

3.2 Q25: assignment within loop

Question 25 asks the students to determine the purpose of a code segment consisting of an assignment statement within a For loop. On each iteration, the loop subtracts an array element dblLoss(i) from a variable dblBalance. When the loop has finished, dblBalance will have had each element of the array subtracted from it. The purpose of the code is thus to compute the new balance after accounting for all of the losses.

I expected that students would perform better on this question, as its structure is simpler, lacking the selection within the loop. In fact their average mark of 1.49 / 5 was somewhat lower than for question 26.

Examples of fully correct answers to this question are

pd004: The overall purpose of the code is to check what is the remaining balance after all the losses have been subtracted from the balance.
pd013: Subtracts the loss amount from the balance amount as long as there is loss amount still available.
pd006: The overall purpose is to remove index zero of loss and following decimos. No matter how long that dblLoss have, it will be totally deducted.
(Like a number of other answers from this offering of the course, this one requires some compensation for difficulties of expression in a language that is presumably not the student's first.)

There are still answers that are almost right, but this question has fewer of these:

pd003: the dblbalance will be deducted until the second end of dblLoss
(Believing, like a number of other students, that looping from 0 to Length–1 omits the last element of the array.)

There were also fewer multistructural answers, answers that describe the process rather than its purpose:

pd051: The code will loop from i=0 to i=dblLoss.Length–1 everytime it loop the value in dblBalance will be minus by the value in dblLoss(i) and the result will be store back to the dblBalance.

There are still wrong relational answers …

pd036: The codes try to sort the number (dblBalance) from the biggest (highest) number to the smallest number
pd041: The overall purpose of this code is:
Subtracting a balance value from a loss that is incurred based on the length of the loss given.

and wrong multistructural answers …

pd002: This code here is to mainly calculate the Balance. According to this balance = balance – loss of (i). Here dblLoss.Length–1 will be the loss subtracted by 1. So i will be 0 to the value that has been subtracted. After finding out (i), we can find out the balance.
pd023: Balance value will be reduced by 1 whenever there is a loss.

However, this question also elicited a number of answers that were so relational as to leave out information that must be considered important …

pd011: Calculate how much money left on the balance

Some of these answers add contextual information that cannot be deduced from the code, putting them into the extended abstract SOLO category …

pd014: To see whether our company still profits or not.
If the value of dblBalance = positive (+) then we get profits
But if it is negative (–), we loss.
pd031: To find out the account balance. To avoid the inefficient fund when the loss is greater than the balance.

Also of interest are the answers that appear not to recognise the loop or the array:

pd016: The code makes the balance deduct from itself
This code is calculate the final balance after deducting the loss.
pd026: It is to calculate the remaining balance amount by the difference of the previous balance and the loss made.

These answers appear to be clearly wrong, but there must remain a shadow of doubt: might it be the case that the student has understood the code correctly but failed to express its iterative aspect?

3.3 Q24: somewhat involved If statement

Question 24 asks the students to determine the purpose of a code segment consisting of an If – Then – ElseIf – Else statement, with a boolean operator in one of the branches. The statement determines whether a permit is needed to remove a tree, based on its health, height, and girth. In more detail, the purpose of the code is to indicate that a permit is required to remove a tree if it is free of disease and either more than 3m tall or more than 0.35 in girth, but that otherwise it can be removed without a permit.

I saw this as the easiest of the three questions. It did not involve loops or arrays, and could be followed by a single pass through the code. Yet the average mark for this question was marginally the lowest of the three, at 1.46 / 5.

It is possible that my expectations for this question were different from those for questions 25 and 26. Section 3.2 mentions answers

that are so relational as to leave out information that must be considered important; yet surely I did just that in the first paragraph of this section, when I wrote that "the statement determines whether a permit is needed to remove a tree, based on its health, height, and girth." In order to demonstrate an understanding of the code, an answer must explain how the determination depends on these factors. Yet such an answer would appear to be tending away from the relational and towards the multistructural, thus confounding the express intent of the question.

Some of the correct relational answers are

pd025: This code performs a check on the attributes of the tree which fulfil the requirements of removing that tree with or without a permit.
If the tree is diseased, then it can be removed without a permit.
If its Height is greater than 3 or the girth of the trunk is greater than 0.35 then permit is required.
Otherwise no permit is required to remove it.

pd044: The overall purpose of this code is the removal of trees that are diseased. If the trees are diseased then no permit is required to remove them. However, if the tree's height is greater than 3 or its girth is greater than 0.35, then a permit would be required to remove the tree otherwise no permit is required.
(This answer is actually wrong in the relational first sentence, but correct in the rest, which explains the code to the required level of detail.)

pd052: It checks which type of tree requires permit to remove. A tree which is more than the height of 3 or the girth is more than 0.35 requires permit. Other than this and diseased tree, no permit is required.
(This is another answer that requires some allowance for difficulties of expression.)

A large number of answers were correct except in confusing *Or* with *And*:

pd002: The overall purpose of this code is to say whether the permit is required to remove the tree or not. When the height of the tree is more than 3 and the girth is more than 0.35; then permit is required, or else if the height is less than 3 and the girth is less than 0.35, then no permit is required to remove the tree. If it is a tree which is diseased also, no permit is required.

pd042: This is a program to check whether a permit is required for removing tree or not. Only if a tree has height more than 3 and girth more than 0.35 and it doesn't have any disease, we will need a special permit to remove that tree. Otherwise, we don't need a special permit.

Other answers showed additional confusion about whether the diseased condition should be combined with *And* or *Or*:

pd016: This code is to check if the tree requires a permit in order to be removed. If the tree is diseased and is above 3m in height and above 0.35m in girth, it requires a permit. If the tree has dimensions lesser than those required or is not diseased, it doesn't require a permit for removal.

pd034: The following code is designed for a permit requirement to remove trees. If the tree is diseased and its height and girth are more than 3 & 0.35 respectively, a permit is then required to remove it. Any other conditions except the above does not require a permit.

pd055: – To protect trees which is still in the save size
– Tall trees is still in protection due to health issues, but when it get to tall (>3 or >0.35) it might get dangerous, so it's permited to be removed.

Many answers completely ignored the disease condition, perhaps because of a failure to understand boolean variables …

pd003: this overall purpose is for someone with dblHeight > 3 or dblGirth > 0.35 will need permit to remove the tree
other than that do not need permit
(This answer also confuses the tree with the feller, a confusion that was not penalised.)

pd051: If the number in dblHeight >3 or the number in dblGirth >0.35 then the messagebox will show the message "Permit required to remove this tree" if not the message box will show the message "No permit required".

One answer is even more suggestive of a failure to understand the boolean variable …

pd009: The code shows Messagebox under different situation. When the Height of a tree is higher than 3 or the Girth of a tree is larger than 0.35 the tree will be Diseased. Other situations there is no permit required.

Probably because the expected answer is more complex than for questions 25 and 26, more answers to question 24 displayed evidence of difficulty with English.

pd049: The purpose of this code is for asking the permit to remove the tree which is under diseases.
According to the code the tree height which is under 3 and the girth which is under 0.35 is no permit required to remove.
However, the tree height is over 3 and the girth is over 0.35 will be required to ask the permit for removing.

pd056: The purpose of the code is to display whether the tree should be removed. When the tree higher than 3 and girth more than 0.35, it is required to be removed. Else if the tree is no higher than 3 and girth no more than 0.35, or the tree got diseased, it can not be removed.
(This answer appears to interpret 'permit required for removal' as meaning 'removal permitted' – or even required.)

This question, more than either of the others, gave rise to relational answers that are not sufficiently detailed …

pd004: The overall purpose of the code is to check whether a permit is required to remove the tree by running through some tests like whether it is diseased or by testing the height and girth of the tree.

pd007: The purpose of this code is to see which condition of a tree are allowed to be freely cut down, without the need of special authorization. The program may be useful when some workers would cut down trees in the city to free up some space. They would not need permission to remove a diseased trees or those that are below certain height and certain girth.
(This is a fine example of an answer in the extended abstract SOLO category.)

pd018: Check whether a permit is required.

pd020: The code are used to determine whether the tree that need to be removed require a permit or not.

pd026: It is to check whether the user requires to have a permit to remove the tree, based on the tree's condition, height, and girth rate.

pd027: To determine if a permit is need to chop down a tree. Using the guidelines given.

Each one of these answers is correct, but all of them could be guessed by reading the code without understanding it. They were therefore not considered satisfactory answers to the question.

4. EXPLORING THE CODE-READING QUESTIONS

In an attempt to explain the disparity in students' marks between the code-explaining questions on the one hand and the code-tracing and code-writing questions on the other, the preceding examination of the three code-reading questions gives rise to a number of questions.

4.1 Understanding vs ability to explain

Some early BRACElet papers suggest that the original intention in using code-explaining questions was to determine whether students could understand code. Indeed, one early paper [14] uses the phrase 'comprehension skills' in its title. This might be problematic: comprehension is a mental state that cannot be measured directly. More recent papers [4] refer to 'the ability to explain code', which is more readily measurable than the comprehension of code.

A more explicit BRACElet goal is to determine whether students can express the purpose of the code in a manner that fits within the SOLO relational category. "A vital step toward being able to write programs is the capacity to read a piece of code and describe it relationally" [14]; "students who cannot read a short piece of code and describe it in relational terms are not intellectually well equipped to write similar code" [5].

Unfortunately, when I set this exam and its marking scheme I had not fully considered the significance of this distinction. In my perception, the students' understanding of a piece of code and their ability to explain that code at a relational level were effectively the same thing, and I believed that by measuring the latter I would be determining the former.

4.2 The marking scheme

The conflation of understanding with ability to explain makes it difficult to allocate part marks to code-reading questions. The marking of these questions has a tendency to all or nothing – full marks for a correct purpose expressed at a relational SOLO level, and few or no marks for any other answer.

Notwithstanding the importance of relational answers as established in earlier BRACElet papers, it is clear that an answer in relational form is far from sufficient: the answer must also be correct. Otherwise on question 26 (section 3.1), marks would be given to pd025, pd026, pd031, pd033, and pd034, even though their answers are completely wrong, simply because they are relational. On question 25 (section 3.2), the same would apply to pd036 and pd041. On any of the three questions, such marks would be allocated to an answer such as "it counts the butterflies in the garden", even though this patently has nothing to do with the code.

On the other hand, correctness alone is also insufficient. A multistructural answer that correctly traces the code fails to display an understanding of it, and so counts for little. An answer such as that of pd020 to question 26 (section 3.1) is really just a translation of the code into English. It demonstrates no understanding of what the code is doing, but merely a skill at reading and interpreting individual statements in the programming language. If the goal is to measure understanding, it is clearly worth few or no marks.

I was not prepared to make the marking scheme entirely binary – full marks for a correct relational answer and no marks for anything else; but I did tend toward such a scheme, with multistructural answers being given at most 2 of the 5 marks. This somewhat defies the standard practice for short-answer questions, which tend to have a clear basis for allocating the full range of part marks to partly correct answers.

4.3 Level of relational detail

It seems that an inherent problem with code-reading questions is the difficulty of specifying the level of detail expected in the answer. Many students fall short of the detail considered necessary; for example, pd011, pd014, and pd031 on question 25 (section 3.2). Others go well beyond the required detail into a full multistructural description of the code. Even after being set practice code-reading questions every week, the students in this class showed very little feeling for the level of detail that was required. It must be remembered, though, that many of the students appear to have ignored those practice questions.

Worse, I did not fully consider when setting the questions that a satisfactory answer for question 24 required a great deal more detail than answers for questions 25 and 26. On question 24 (section 3.3), the answers of pd004, pd007, pd018, pd020, pd026, and pd027 appear to be at or beyond the relational level of correct answers to questions 25 and 26, yet do not demonstrate a full understanding of the intricacy of the code.

Perhaps the best illustration of underspecified relational answers is pd018 on question 24 (section 3.3). This student, who wrote one of the least specific answers for this question, scored full marks for the other two reading questions and 89% on the exam. The student surely understood the code – but did not understand how that understanding was to be expressed.

It might be argued that any correct relational answer, no matter how short on detail, should be considered correct. But almost anyone, even somebody with no knowledge of programming, might be able to guess a generic purpose of permits for tree removal; and therefore an answer expressing that purpose cannot be taken as showing an understanding of the code.

Taken to the logical extreme, a correct relational answer to any of these questions would be "this code does what it was designed to do". Yet once again, such an answer shows no understanding whatever of the code, and so should receive no marks, notwithstanding that it is both relational and accurate.

In retrospect, question 24 is simply a bad question. Given the intention of having three virtually identical questions without iteration or recursion, one for tracing, one for reading, and one for writing, it was felt that some complexity was warranted, and the form chosen for that complexity was the somewhat involved If statement. However, I now believe that the form is quite inappropriate for a code-reading question that expects a relational answer. In terms of cognitive load theory [10], it places a very

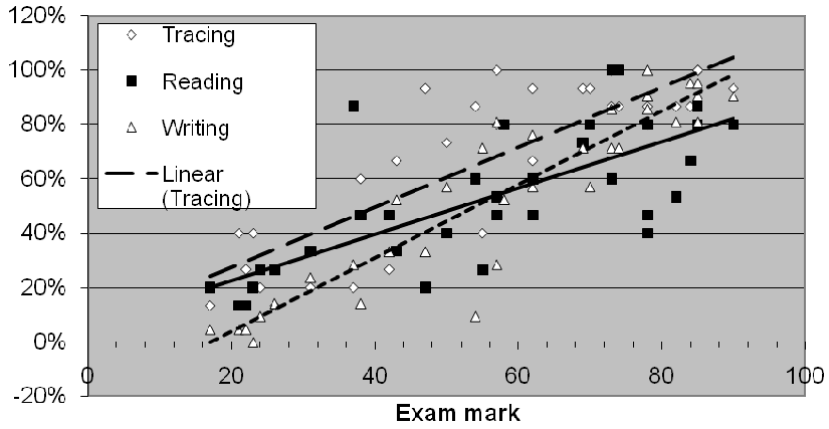


Figure 2: Later students' marks in tracing, reading, and writing, against overall exam mark

high load on working memory, a load that is almost certainly inappropriate for a short examination question.

4.4 Problems with expression

The Leeds working group used only multiple-choice questions. "We were concerned that if students were required to explain the function of a piece of code, poor performance might be due to a lack of eloquence, not a lack of understanding of the code" [3]. In its early days, the BRACElet project adopted the same practice, although possibly not for entirely the same reason: "We decided to maintain the multiple-choice question (MCQ) format used by the Leeds group in our study in order to ensure that we had no variation in our data analysis between markers and institutions" [12]. However, "Our problem set was further extended to include a set of questions that were more open, subjective questions designed to test a higher level of the Bloom taxonomy. An example of a more open question is a question that asks the student to explain the purpose of a code snippet" [12].

It would seem from a handful of the answers examined here that eloquence, or perhaps written expression at a more basic level, might indeed be an issue. Most of the answers were comprehensible, notwithstanding that many of them were written by students whose first language is not English. A few, such as pd049 and pd056 in question 24 (section 3.3) and pd003 and pd006 in question 25 (section 3.2), were less clear. However, this problem need not necessarily concern us. Many other disciplines require a great deal more writing than a computing degree; and regardless of the discipline area, a degree from an English-speaking university implies a certain facility with written and spoken English, and students who lack that facility will not tend to earn marks as high as those who possess it.

It is thus possible that a student who understands the code can produce an answer that is confusing or just plain wrong because of a lack of fluency in English; but this can be considered a problem with the student's language skill rather than a problem with the question type.

5. A SUBSEQUENT DATASET

For a subsequent exam on a different campus I used the same questions but revised the marking scheme, giving full marks to a correct relational answer and three of the five marks to a correct multistructural answer. Figure 2 shows the students' marks on the reading, explaining, and writing questions. There is no longer a significant difference between question types. The students' tendency to respond multistructurally rather than relationally to code-explaining questions, now just a reduced likelihood of scoring full marks on those questions, shows as a shallower slope to that trend line. Even with that tendency, I suspect that the line would have been less shallow had question 24 not been such a bad question.

Discussions with members of the BRACElet project suggest that this sort of marking scheme is in fact the norm, and that my own almost binary marking scheme was very much the exception.

6. DISCUSSION

Code-reading assessment can help to address important research questions, to which the answers might eventually have major impacts on the way programming is taught.

In addition, the students' answers to the code-reading questions are extremely illuminating in a number of different ways. In some cases, at least, they offer explanations for students' weaknesses that are not evident from other types of exam question.

However, there do appear to have been two problems with my use of these questions, one of which is easily remedied, the other of which is perhaps not so easily dealt with.

The problem of the marking scheme potentially attends any new type of question proposed for assessment. (This is not to say that the marking of old question types is appropriate; just that it tends to conform to some sort of accepted standard.) The simple solution is that when a new question type is proposed, a marking scheme should be proposed along with it. My adoption of code-explaining questions is evidence that when a new question type is proposed without a marking scheme, a teacher of considerable experience can devise a scheme that simply doesn't work.

In response to this problem, I propose in Appendix B a fairly generic marking scheme for a 5-mark code-explaining question. If other teachers using and reporting on such questions report their

own marking schemes. some notion of a standard marking scheme might eventually be arrived at.

The problem to which I currently have no answer is that of the level of relational detail required in an answer. This is a critical aspect of the answer: too little detail and the answer might well be a guess; too much detail and the answer might well be a multistructural translation of the code into English. In this particular course a reasonable effort was put into preparing the students, both with the weekly exercises and with the wording of the question preamble, but this effort appears not to have paid off.

It might be worth considering preceding code-explaining questions with a complete example, question and answer, on the examination paper. This should help if the students' problem is not knowing what level of explanation is required; it would not help if their problem is a genuine inability to explain the code at that level.

It might also be worth considering multiple-choice code-explaining questions. Before my experience with this exam I might have thought it impossible to devise plausible distractors for such questions. However, students' answers are often a good source of distractors, and I could now, for example, propose the following choices for question 26:

(a) To find the longest title in the array of titles.

(b) To find the length of the longest title in the array of titles.

(c) To move the longest title in the array of titles to the first place in the array.

(d) To sort the array of titles according to title length.

(e) To find a specified title in the array of titles.

It would be particularly interesting to include both free-form and multiple-choice code-explaining questions in the same examination paper and to explore the relationship between the marks awarded for each.

There is a clear value to code-explaining examination questions, both as a research instrument and as an instrument of assessment. But I believe that if they are to achieve their full potential in assessment, a way must be found of ensuring that the students understand just what sort of answer will achieve full marks.

7. ACKNOWLEDGMENTS

I am grateful to the BRACElet leaders, Raymond Lister, Tony Clear, and Jacqui Whalley, for initiating the project, for developing it, and particularly for publishing their research design and inviting other researchers to implement it. I am also grateful to the BRACElet members on whose examination questions some of my own were based. Discussions with members of the BRACElet project and with people attending Koli Calling 2009 have helped to reshape this paper considerably. For this, too, I am grateful.

8. REFERENCES

[1] JB Biggs and KF Collis (1982). Evaluating the quality of learning: the SOLO taxonomy (Structure of the Observed learning Outcome). Academic Press, New York.

[2] T Clear, R Lister, Simon, DJ Bouvier, P Carter, A Eckerdal, J Jacková, M Lopez, R McCartney, P Robbins, O Seppälä, and E Thompson (2009). Naturally Occurring Data as Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. SIGCSE Bulletin 41(4).

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

[4] R Lister, C Fidge, and D Teague (2009). Further evidence of a relationship between explaining, tracing, and writing skills in introductory programming. 14th Annual Conference on Innovation and Technology in Computer Science Education, Paris, France.

[5] R Lister, B Simon, E Thompson, JL Whalley, and C Prasad (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. Bologna, Italy, 118-122.

[6] M Lopez, J Whalley, P Robbins, and R Lister (2008). Relationships between reading, tracing and writing skills in introductory programming. 2008 International Workshop on Computing Education Research (ICER'08), Sydney, Australia, 101-111.

[7] M McCracken, V Almstrum, D Diaz, M Guzdial, D Hagen, Y Kolikant, C Laxer, L Thomas, I Utting, and T Wilusz (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bulletin 33(4), 125-140.

[8] J Sheard, A Carbone, R Lister, B Simon, E Thompson, and JL Whalley (2008). Going SOLO to access novice programmers. 13th Annual Conference on Innovation and Technology in Computer Science Education, Madrid, Spain, 209-213.

[9] Simon, M Lopez, K Sutton, and T Clear (2009). Surely we must learn to read before we learn to write! 11th Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, 165-170.

[10] J Sweller (1988). Cognitive load during problem solving: Effects on learning. Cognitive Science 12, 257-285.

[11] A Venables, G Tan, and R Lister (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. Fifth International Workshop on Computing Education Research, Berkeley, CA, USA, 117-128.

[12] J Whalley, T Clear, and R Lister (2007). The many ways of the BRACElet project. Bulletin of Applied Computing and Information Technology 5(1).

[13] JL Whalley and R Lister (2009). The BRACElet 2009.1 (Wellington) specification. 11th Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, 9-18.

[14] JL Whalley, R Lister, E Thompson, T Clear, P Robbins, PKA Kumar, and C Prasad (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. 8th Australasian Computing Education Conference (ACE2009), Hobart, Australia, 243-252.

APPENDIX A: The three sets of equivalent questions

These are the nine questions written to explore students' capability in comparable tracing, reading, and writing tasks. On the exam paper the three tracing questions came first, followed by the three reading questions, and then the three writing questions. Here they are grouped according to the code structure rather than the task type, to highlight their comparability.

The language used in this course is Visual Basic. Java programmers might need to adjust to the absence of braces and to the convention of prefixing variable names with a 1-letter or 3-letter indication of their type.

1. A somewhat involved If – ElseIf – Else statement

Tracing version

Q21. Suggest a value for intAge and a value for strDay that will result in a value of "Half price" for strPrice when the following code statements are executed.

```
If intAge <= 6 Then
   strPrice = "Free entry"
ElseIf intAge <= 15 And
        strDay.IndexOf("S") = 0 Then
   strPrice = "Half price"
Else
   strPrice = "Full price"
End If
```

Expected answers: any integer from 6 to 15 inclusive and any string beginning with "S".

Reading version

Q24. Explain what the following code does. You are not being asked to explain each line of the code; you are being asked to explain its overall purpose.

```
If blnDiseased Then
   MessageBox.Show("No permit required")
ElseIf dblHeight > 3 Or
         dblGirth > 0.35 Then
   MessageBox.Show("Permit required to
         remove this tree")
Else
   MessageBox.Show("No permit required")
End If
```

Expected answer: It indicates that no permit is required to remove a tree if it is diseased, or if it has a height <=3 and a girth <=0.35; otherwise a permit is required.

Writing version

Q27. The movie price calculator form below [the form was illustrated] shows 3 textboxes, *txtBasePrice*, *txtRating*, and *txtAge*, that accept the normal or base price of a ticket, the rating of the film, and the age of the customer. A button *btnCalculate* executes code that will display the actual price for this ticket in another textbox, *txtThisPrice*.

The price of a ticket is the base price multiplied by a multiplier, where:

- for customers aged under 6 the multiplier is zero;
- for customers aged over 64 the multiplier is 0.75;
- for customers aged from 18 to 64 the multiplier is 1;
- for customers aged from 6 to 17 the multiplier is 0.5, unless the film rating is "R", in which case the multiplier is 1.

Write code to calculate the ticket price and display it, appropriately formatted, when *btnCalculate* is clicked. You are not expected to reproduce the parameters (the items in parentheses) for the event handler. Just write "(parameters)" where they would normally appear. You may assume that valid data has been entered in the first three textboxes – your program is not required to check this.

If statement from expected answer:

```
If intAge < 6 Then
   dblMultiplier = 0
ElseIf intAge > 64 Then
   dblMultiplier = 0.75
ElseIf intAge < 18 And
         txtRating.Text <> "R" Then
   dblMultiplier = 0.5
Else
   dblMultiplier = 1
End If
```

2. A simple loop with assignment

Tracing version

Q22. What will be the value of strIng1 after the following code statements are executed?

```
Dim i As Integer
Dim strIng1 As String = ""
For i = 1 To 4
   strIng1 = strIng1 & CStr(i * 2) & " "
Next
strIng1 = strIng1 & Environment.NewLine
        & "When do we capitulate?"
```

Expected answer: "2 4 6 8
When do we capitulate?"

Reading version

Q25. Explain what the following code does. You are not being asked to explain each line of the code; you are being asked to explain its overall purpose.

```
For i = 0 To dblLoss.Length - 1
   dblBalance = dblBalance - dblLoss(i)
Next
```

Expected answer: It subtracts every loss (or every element of the array dblLoss) from the balance.

Writing version

Q28. A form has a textbox *txtNum* in which the user has entered a positive integer. Write lines of code to calculate the product of the numbers from 1 to the integer in *txtNum*, and display that product in a message box. (A 'product' of numbers is what you get when you multiply those numbers together.)

Your code does not need to be a full event handler; just write the lines that do the task described, like the code samples in questions 21-26. And your code is not required to check whether the textbox contains a positive integer – just assume that it does.

Expected answer:

```
n = CInt(txtNum.Text)
intProduct = 1
For i = 2 To n
   intProduct = intProduct * i
Next
MessageBox.Show(intProduct, "Q28")
```

3. A simple loop with If and assignment

Tracing version

Q23. What will be the value of intLast after the following code statements are executed?

```
Dim intNums1() As Integer = {1,5,2,4,2}
Dim intNums2() As Integer = {4,2,4,7,1}
Dim intLast As Integer = -1
Dim i As Integer = 0
Do
   If intNums1(i) < intNums2(i) Then
      intLast = i
   End If
   i = i + 1
Loop Until i = intNums1.Length
```

Expected answer: 3

Reading version

Q26. Explain what the following code does. You are not being asked to explain each line of the code; you are being asked to explain its overall purpose.

```
strOne = strTitle(0)
For i = 1 To strTitle.Length - 1
   If strTitle(i).Length > strOne.Length
                           Then
      strOne = strTitle(i)
   End If
Next
```

Expected answer: It finds the longest title in the array of titles.

Writing version

Q29. A program has an array of integers called *intChoice*. Some of the integers in the array, but not all of them, will have values of zero. Write program code (just the relevant lines, not an event handler) to count and display the number of non-zero integers in *intChoice*.

Expected answer:

```
iNumber = 0
For iCount = 0 To intChoice.Length - 1
   If intChoice(iCount) <> 0 Then
      iNumber = iNumber + 1
   End If
   Next
MessageBox.Show(CStr(iNumber), "Q29")
```

APPENDIX B: Proposed marking scheme for Question 25, worth 5 marks

Relational (summary) answers

Answer that provides a correct relational summary, with or without a line-by-line description: 5 marks.

Answer in summary form, correct but incomplete, eg "subtracts the loss from the balance": 2 or 3 marks depending on the extent of the incompleteness.

Summary answer that's possibly correct but too general, eg "it balances the books": 1 mark.

Summary answer that's completely incorrect, eg "subtracts 1 from each balance" or "it finds the sum of all the balances": 0 marks.

Line-by-line (multistructural) answers

Fully accurate and complete answer couched as a line-by-line description: 4 marks.

Partially accurate answer in line-by-line form: proportionally less than 4 marks.

Praxis-oriented teaching via client-based software projects

Malgorzata Mochol
Freie Universität Berlin, Institute for Computer Science, Networked Information Systems
Königin-Luise-Str. 24-26, D-14195 Berlin
mochol@inf.fu-berlin.de

Robert Tolksdorf
Freie Universität Berlin, Institute for Computer Science, Networked Information Systems
Königin-Luise-Str. 24-26, D-14195 Berlin
tolk@ag-nbi.de

ABSTRACT

How to work in teams, manage people, negotiate with clients, delegate tasks, plan milestones, deliver results on time and, in the end, conduct a "real" IT project are the aims of our client-oriented IT project for bachelor and master degree students. In this paper we explain the ideas behind practice-oriented IT projects as well as the experiences and lessons learned from past projects.

Categories and Subject Descriptors

K.3.2. [**Computer and Education**]: Computer and Information Science Education—*computer science education, information systems education*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*management techniques (e.g., PERT/CPM), systems analysis and design*

1. EDUCATIONAL OBJECTIVES

Student IT projects at universities, even if they are course requirements of a computer science degree[4], are usually internal courses characterized by invented (fictional) problems with well-defined goals and requirements, default and constant workflows, predefined work packages and coding from scratch without the need to understand foreign code. According to[6], the traditional computer science curriculum leads to certain difficulties, from the education aspect to the problems students encounter in the working world following graduation; these courses are marked by their lack of practical relevance to actual industrial IT projects with which graduates are confronted when they start their working life. To enhance the student's skill set, we have developed a concept at the Institute for Computer Science[1] of the Free University of Berlin[2], for IT projects with active participation from industry partners. Our main teaching goals in practice-oriented projects are:

- to simulate real-life situations with regard to real-world problems, industrial clients, proposals, contract, milestones and the final product hand-over;
- to give our students a team-based experience while creating a comprehensive real software product[3] for a real-world application;
- to give our students the opportunity to work on their communication, processing and leadership skills: problem solving, conflict management (between client and team members), soft skills, personnel management, which are, according to[2], *"the critical tools essential to resolving and taking action on practical problems within families, workplaces and communities"*;
- to give project participants the opportunity to learn about risk- and crisis management;
- to teach the students how to set up internal and external meetings, including preparation of productive meetings and a conducive meeting atmosphere;
- to afford our students the opportunity for project-based learning in order to develop/enhance the ability to arrive at informal judgment, the capability to address a specific problem in complex, real-world settings, their communication skills as well as technical competence.

2. CONCEPT AND CLIENTS

During our software courses, students are involved in industry-like projects which require hands-on participation. Our commercial partners that usually have already collaborated with our group in the context of research projects, events and other activities, play the role of a real-world commercial client (different partners in each semester).

The project is entirely managed by the students, i.e. the students have (almost) full control of the work flow, its progress, time plan as well as being held responsible for all project-related ups and downs. The participants negotiate and make arrangements with the client, interview the client in order to conduct a requirements analysis, prepare a proposal based on the client's needs, plan milestones and deliver them on time and to the satisfaction of the client.

In past semesters our courses were conducted in collaboration with local and regional firms in Berlin and the state of Brandenburg and, starting this summer term, also with international organizations:

[1] `http://inf.fu-berlin.de/en/index.html`
[2] `http://fu-berlin.de`

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

- For *neofonie GmbH*[3] students developed a web-based tool for editing, application and evaluation of the Hearst Patterns[5] (summer term 2007).
- With *Condat AG*[4] students worked on a metadata-based repository for saving and intelligent querying TV-channel data. Specifically, the students developed a web-based application with an RDF-metadata repository containing TV-program data (including the transformation of the data into RDF format) and a corresponding API to store TV data in and query from the repository (winter term 2007/2008).
- For *Projektron GmbH*[5] students designed and implemented a semantic search engine for tickets within the client's project management system; the application developed analyzed existing tickets and identified their commonalities so as to, in the case of similar tickets, mark them as cognated tickets (summer term 2008).
- For *Espresto AG*[6] students utilizing modern web technologies (e.g. AJAX) enhanced an existing content management system with regard to usability and clarity issues while keeping in mind browser compatibility, which was vital to the client (winter term 2008/2009).
- During summer term 2009 we were cooperating with *Village Scribe Association (VSA) vzw.*[7], an association for the advancement of innovative information and communication technologies for rural development in South Africa. The students worked on awareNet[8] extensions, an interactive education and communication software featuring an intelligent platform that provides semiautomatic monitoring to protect its users. Participants developed an image-uploader and working on a lexical analysis for the awareNet text fragments.

3. ROLES WITHIN THE PROJECT

As mentioned in the previous section, the project is led by students: they act as contractor and are responsible for the success (or failure) of the project. The students with their lecturers and commercial partners play different roles that simulate real-world projects (cf. Fig 1):

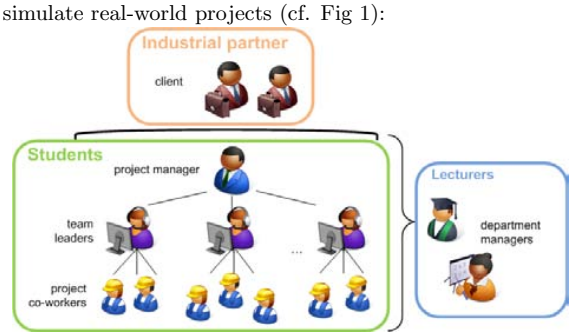


Figure 1: Different roles in the project

[3] http://neofonie.de
[4] http://www.condat.de/
[5] http://projektron.de/
[6] http://www.espresto.de
[7] http://dorfschreiber.org/Intro.html
[8] http://www.awarenet.eu/

Project manager(s) At the beginning of each semester students, usually graduate (master) students, apply for one or two project management positions. They send a CV with a letter of motivation of why they want and are suited for the position. During the course of the project the manager deals with client's various requests, learn to negotiate with clients, manage the problems in the project team, which is usually composed of 20-25 students, learns to cope with major technical issues, delegates and coordinates work, maintains an overview of the project's tasks and activities, and motivates the team. The project manager is responsible for the proposal, intermediary and final presentation of the results, and success of the overall project. The project manager forms teams appropriate to the project requirements, designates team leaders (3-5 team leaders depending on the project settings) and, at the end of the semester, assesses the team leaders and together with them prepares grading proposals.

Team leaders After the project management has been appointed, it selects the team leaders. Leaders learn how to manage a small team of 3-5 persons and delegate the work to their team members. They must coordinate and delegate the tasks, maintain a careful overview of the work progress, carry out development work in order to be able to discuss the technical issues and, in some cases, persuade the team to take on a particular solution. Team leaders motivate the team members, settle a dispute, if necessary, and report to the project manager on the project's progress. At the end of the semester team leaders evaluate and assess each team member and together with the project manager prepare the grading proposal.

Project co-workers Co-workers work on the implementation/technical level and are responsible for their assigned tasks. They report to the team leader with regard to the task's progress and difficulties.

Client Each semester one of our commercial partners act as a client. The client presents a dilemma and specifies a number of issues and requests to be solved by the students, however, as in real life, the client does not provide the problem's exact specifications. The client decides whether to accept the proposal submitted by the students, the suggested milestones and, at the end, whether to approve of the final results.

Lecturers Prior to the project the lecturers get together with the commercial partner to prepare the project framework: define the project area and goals, check the complexity of the tasks, clarify the project volume and review the technologies which can theoretically be utilized to solve the problem in question. The lecturers supervise the project, specifically by lending support and assistance to the project managers and team leaders. We introduce the students to project management and coach them mainly at the very beginning of the semester. However, most important of all we remain in the background and let the students learn by doing and experience the ups and downs of a real-life project. We only intervene if we think the situation urgently requires our direct intervention.

4. RUNNING THE PROJECT

In the following, we outline the main steps and flow of the project shown in Fig. 2:

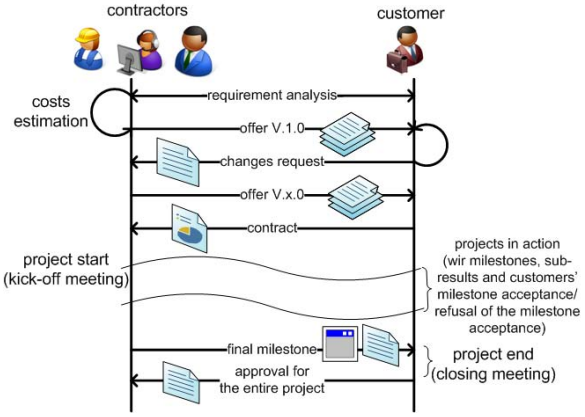


Figure 2: Main steps within the project

Before the kick-off: The students are in a constant dialog with the client, carry out the requirement analysis, prepare a detailed schedule with milestones, describe sub- and end results, and submit it to the client, who will decide whether to accept, reject or call for revision.

After the kick-off: If the students win the project they work according to the plan they prepared. After the project kick-off-meeting, during which the proposal is signed by the client and the student project manager, the implementation of tasks, goals and milestones starts.

Internal and external communication: To organize the internal project communication students use e-mails as well as project and tasks mailing lists. Furthermore, to establish software development process and policies, and to keep an overview of the project team's progress, we recommend that the students utilize Trac system[9], which is a wiki- and issue-tracking system for software development projects. Trac uses a minimalistic approach to web-based software project management. In the last summer term, the Trac ticket system was especially used for sub-tasks allocation: the project manager and the tasks leaders assigned tickets with a specific sub-task to a particular individual (assigned ticket), the individual accepts the ticket (active ticket) and after completion of the task closes the tickets (closed ticket). The utilization of a project management system allows students to become familiar with the activities that are common in IT practice and to give them a chance to learn aspects which a real-world project manager or team leader has to deal with.

External communication with the client is almost exclusively conducted by the project manager who is also responsible for keeping the client (as far as necessary) up-to-date on the project's progress and for the on-time delivery of the milestones. Only in special cases, where both the project manager and the client decide that direct communication between team members and client is needed, e.g. to clarify technical issues, specific students contact their counterpart on the client's side.

[9] `http://trac.edgewall.org/`

Circumstances for a real project: To realistically simulate the problems and difficulties of a real-world project the client treats the student as if they are contractors, i.e. the students can expect no special treatment. The clients (i) may react harshly if they are not pleased with the progress of the project, (ii) may demand more than what was originally agreed in the contract in order to challenge the students' negotiating abilities, (iii) might sometimes change some parts of the tasks or bring forward the deadline for milestone delivery to stimulate the students' resourcefulness.

Project close out: If the project is brought to a successful conclusion, the students will be rewarded not only with good grades and ECTS points but usually with a prize. For the current summer term, for instance, the client Village Scribe Association vzw. was so pleased with the results achieved that it asked the South African embassy in Germany to host the closing ceremony of our project. Embassy officers and staff and other members of the VSA was attending the event as well. At the ceremony, the students presented the goals and final results of the project with a demonstration of the software application developed.

5. LESSONS LEARNED

Since the launch of the practice-oriented student software projects, the students' activities along with their motivation, project progress and coordination, communication within the team and with the client have been closely monitored. On the basis of our observations we have discovered a number of recurring characteristics and issues:

- There is an initial period of "disorientation" among the students which eases up around the third project week.
- We have noticed that the view "success of the project stands or falls with the project manager" is even more evident in student projects where there are individual team members whose morale crashes, even if the motivation of the rest is constant.
- Communication is central to the project: even if internal communication, primarily at the start, leads to difficulties, they often dissipate over the course of the project. Here again, the project manager plays a crucial role.
- Clear briefing and goals are crucial to the project's success and the students' morale. Even if goals are not very well defined at the beginning of the project, it is extremely important that after the requirements analysis and contract signing *all* participants (i) share the same view with regard to the problem to be solved, (ii) know the goals, and (iii) hold a common vision of the solution.
- Students are quite knowledgeable with regard to software development but usually have little understanding of project management issues and team work.
- Students have reservations regarding learning and trying out new technologies. For instance, in the current project, many discussions and difficulties arose with regard to Python and its frameworks due to the fact that most students were familiar only with Java or C++.

- There are always 1 or 2 students who drop out in the middle of the project; this can discourage the remaining participants, but, on the other hand, it has a positive impact on their learning and experience in the context of risks management.

- Participating students and especially students playing leadership roles are highly motivated since they are working for a "real" client with real-world requirements, and they get to experience the rewards of meeting an actual need with the software solution they themselves develop; these observations overlap with the findings presented in[6].

- The lecturers should, to some extent, monitor the communication between the partners and the work within the project. We do so by requiring all participants to send weekly reports with their achievements during the week, problems encountered and their plan for the following week. Searching for common problems and addressing them with the project management and team leaders seems to be an appropriate balance between tight supervision, which is opposed to our concept, and letting the students run completely without advice.

  Especially in the starting phase, the lecturers have to insist that proper communication and management structures and practices should be established (e.g. no issues are left open without assigning a person in charge and a deadline).

6. CONCLUSIONS & DISCUSSION ISSUES

Our real-world business-oriented project courses are highly popular and beneficial to all parties. Due to the huge demands from both the students and the companies we have not been able to accommodate everyone. Even if some universities have already conducted similar courses[1, 8, 9] this is in our opinion still not enough and therefore more universities should offer such praxis-oriented software projects considering them as an important issue in the students education. This, in turn, would (i) enhance the qualifications of students while strengthening commercial cooperations - students familiar with real-life problems prior to their graduation are preferentially sought by industry[8], (ii) strengthen not only the students' programming skills but also soft skills - industry is emphasizing the need for graduates to have both technical skills and soft skills[7] and (iii) strengthen or establish binding between the university and the industry[9]. However, despite the obvious benefits of such projects and the positive feedback from both students and companies, we still have reservations regarding the format of the course and how projects are implemented.

- **Do we conduct software development for free?** Students develop and turn over to a client a working application (software with source code, documentation and user handbook), i.e. we give away free software which the company uses for its own while we retain no user rights. Students receive no remuneration for their many hours of work.

- **Continuity?** Until now, the role of the client has been played by companies that change with each project. One question worth considering is whether it would be better and more beneficial for students if (i) the client's relationship (and perhaps also the subject's relationship) to a group of students remained constant over the course of the study, so that (ii) students could build up a sustainable relationship with one company (e.g. resulting in a related master/bachelor's thesis, student jobs or a permanent position after graduation).

- **Evaluation?** Until now, the observations outlined in Sec. 5 remain just that and benefits may be gained from a structured formal evaluation approach (systematic evaluation of learning outcomes, student feedback, project management, etc.).

- **Overall Strategy?** There should be a defined overall strategy of the teaching institution on how such a course is embedded in further industry-related activities like students internships in companies, research cooperations with industry, or finding industry partners for invited talks in regular courses.

7. REFERENCES

[1] K. Alho. Using the world wide web to assist software project course work. *Information & Software Technology*, 40(4):245–248, 1998.

[2] H. Boggs and S. Laurenson. Problem-based teaching: A bridge to meaningful learning. Ohio State Univ., Center on Education and Training for Employment, 1997.

[3] John F. Dooley. A software development course for cc2001: The third time is charming. In *Proc. of the 13th Annual Conf. on Innovation and Technology in Computer Science Education*, pages 346–346, 2008.

[4] K. A. Hawick H. A. James and C. J. James. Teaching students how to be computer scientists through student projects. In *7th Australasian Computing Education Conference (ACE2005). Conferences in Research and Practice in Information Technology(CRPIT)*, pages 259–267. Australian Computer Society, Inc., 2005.

[5] M.A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proc. of the 14th Internat. Conf. on Computational Linguistics*, pages 539–545, 1992.

[6] C. Johansson and Pc Molin. Maturity, motivation and effective learning in projects - benefits from using industrial clients. In *Proc. of the 1995 2nd Internat. Conf. on Software Engineering in Higher Education*, pages 99–106. Computational Mechanics Publ, 1995.

[7] D. Rundus K. Christensen and Z. G. Prodanoff. Partnering with industry for a computer science and engineering capstone senior design course. In *Proc. of the ASEE Southeast Section Conference*, 2003.

[8] C. 0. Ruud and V. J. Deleveaux. Developing and conducting an industry based capstone design course. In *Proc. of the 27th Annual Conf. Teaching and Learning in an Era of Change*, volume 2 of *Frontiers in Education Conference*, pages 644–647, 1997.

[9] V. Korhonen V. Isomöttönen and T. Kärkkäinen. *Agile Processes in Software Engineering and Extreme Programming*, volume 4536/2007 of *LNCS*, chapter Power of Recognition: A Conceptual Framework for Agile Capstone Project in Academic Environment, pages 145–148. Springer, 2007.

Exploiting the Advantages of Continuous Integration in Software Engineering Learning Projects

Sandro Pedrazzini
SUPSI, University of Applied Sciences
of Southern Switzerland
6928 Lugano-Manno, Switzerland
Canoo Engineering AG,
Kirschgartenstrasse 5
4051 Basel, Switzerland
sandro.pedrazzini@supsi.ch

ABSTRACT

There are some practices in software development that are hardly exercised at the university. Some of them should be introduced in the courses or in the labs, not only because they will be useful in the working world, but because they add benefits to the learning process itself. One of those practices is continuous integration, a state-of-the-art software engineering methodology. In this paper I will explain how we setup a continuous integration platform for our students, introducing continuous integration practices as part of our project-based software engineering lab, and what benefits can students achieve from it.

Keywords

Continuous integration, project-based learning, team work, active learning, responsibility.

1. INTRODUCTION

The core topic of the whole process and experiments is the setup and regular use of a so called "Continuous Integration Platform", i.e. a platform where the students can access and use tools for version management and continuous integration (CI). The platform is used by the students to manage their projects and to exercise agile development methodologies during their works. The concrete aspects of such a recurrent and continuous practice match well with project-based learning (meant as a group activity that goes on over a period of time, resulting in a product), and helps to get more benefits from it. In project-based learning you typically have a timeline and milestones, and the means to evaluate your improvements ([11]). Correctly using CI tools allows to keep the software project under control but also to keep track of it, being able to better evaluate the project itself and the personal improvements.

Exercising such methodologies, strongly iterating over all development phases during the whole project timeline, with short analysis and development cycles, already helps in better understanding the advantages of the agile way of development.

But this is not all.

Observing the students' different approaches to the projects allowed us to understand, besides this, some more learning benefits of exercising such a practice in our projects.

2. CONTINUOUS INTEGRATION

Integration is a practice in software development that happens when in a development team, a member of the team must integrate his work with the work of the other colleagues, or when the work of an entire team must be integrated with a series of external modules, production environments, databases, configurations, etc., specific to a particular platform. Often the integration is performed as the last development step before going into production. Because it is a very critical phase in the project, it may require days, weeks or even months.

These days, weeks or months are the periods of greatest pressure for the development team right because they correspond to the phase prior to the production. Every day is a day more of delay in putting the application into production. Each error found in this phase is likely to be managed and resolved in a hurry, without the necessary care and quality.

The general idea behind the practice of continuous integration is to transform the process of integration into a non-problematic event, or, even better, to turn it into a "non-event" at all.

How to achieve this goal? Integrating the most frequently as possible, automate the verification needed during the integration, executing all unit and functional tests as part of the process and always generating the final project artifacts (deliverables).

Continuously integrating means integrating at each interaction, after any changes that a developer executes in the central code repository (common repository to which all team members have access and where everyone sends his changes). This means that at each new integration every developer adds only a few hours project work. So, eventually discovered integration problems can only be related to such few hours of work and are therefore easily identifiable.

The automation of the integration process is permitted by the combination of some tools: a version control system (VCS, some products available: CVS, [2], Subversion, [6]), a build tool (like

Ant, [1], Maven, [5] or others) and a system (like TeamCity, [7]) that regularly monitors the changes and triggers a build process at each modification.

So, as soon as a developer performs a change to the code repository (where all the project data are stored), sending his own last changes to the central server running a VCS, another process, through polling, notices the changes and starts a build process.

With build process we mean the full compilation, the execution of all tests, the generation of distribution files ("deliverables"), and any further control, such as dependencies checking, style checking, etc.

3. THE PLATFORM

Along with the rest of the documentation of the course of software engineering and with the main book used in the course ([12]) we set up a platform for continuous integration, used during projects, accessible through the internal e-learning platform already used for the various learning stages of the course.

We integrated an existing product ([7]) and provided a dedicated configuration in order to facilitate its use for internal students' projects.

In particular, we developed a basic script (template) for the build, so that each project can be recognized by the CI tool and treated uniformly (compilation, test execution, generation of deliverables, different kinds of checks) without the need for further special configurations.

The script has been developed following the general needs of our projects, but can be freely modified and adapted by the students at any time. Each team manages its own copy.

The infrastructure used allows the students to remotely monitor the project build process in different ways: through a Web interface, through email notification (various levels of configuration are possible), or through a plug-in installed on the suggested development environment (Eclipse IDE, [3]).

3.1 General Positive Values

Here is a list of general positive values that such a CI platform can bring to the students' activity on projects:

- quick feedback on the situation of the project (development, problems, regression testing, functional testing, etc..),
- experience on all activities and processes of an end-to-end project, already from the first iteration,
- access the history of the whole project (past changes, old failing tests subsequently corrected, decisions on design and redesign, refactoring, etc.),
- remotely monitor the progress of various projects, tests, changes, etc. (this is especially useful to the teacher),
- ensure a quick build (because made with a system of agents that provide free resources) even in the presence of many projects and many changes,
- administer the project in a simple and intuitive way, thank to a rich Web 2.0 interface.

4. DIDACTICAL VALUES

More than the general CI positive values, already generally recognized as added values during the software development process, there are some more special values for students, related to their activity and their project-based learning process.

4.1 Active learning

Accessing the continuous integration tool through the e-learning platform allows the students to administer their projects in an active way, being able to monitor their single project evolution, its maintenance, its tests and the problems related to them.

4.2 Tracking personal improvement

From the information made available by the CI tool the student can derive his personal improvement over time, looking at the decisions taken regarding the evolution, the project growing, the functionality implemented, the test coverage, the failing tests corrected, etc.

4.3 Work in team

The continuous integration platform facilitates and promotes working in team, an essential aspect of software development.

Each team member can send his changes separately and see them automatically integrated into the final product. Each member can also monitor the execution of tests (directly through the Web, through the available plug-in or through a notification sent by mail by the CI tool). The work of each member can be accessed and changed by all other members of the team (collective ownership). Code reading, code understanding and code reviewing become part of the usual work for every team member.

4.4 Responsibility

The platform, as used in the course, forces each element of the team to take his responsibility for correcting possible errors (test failed) due to his last modification.

Whoever takes care of the correction can register himself into the platform, and the latter will inform the rest of the team that someone has taken the responsibility of the correction. The team will also be informed as soon as the correction will be effective.

The direct consequence of this way of working is that every student, before sending the changes to the central repository, must check on his local machine if all tests are passing. The student is forced to pay attention to the quality of the code, increasing at the same time the degree of accuracy.

Responsibility is, by the way, a crucial attitude in project-based learning. Exercising it through the project itself can have a positive impact on improving each single student's learning process.

4.5 Maintenance

There is a phase in the software development life cycle that can hardly be exercised at the university: this is the maintenance.

There are different visions on maintenance, and definitions slightly vary depending on whether we look at it in terms of traditional methodologies or in terms of more agile ones ([13]). In both cases, however, maintenance is seen as the phase that follows the first delivery.

In the academic world this step is often overlooked because what counts more in exercises and students' projects is usually the delivery. This is in contrast to what happens in the software industry, where the maintenance phase is a crucial one, because it takes years and determines the further development and success of an application.

Working with a continuous integration platform allows the students to transform the phase of development to a phase of maintenance. The early integration of work and the end-to-end project setup lead to the ability to deliver (even partially) in faster cycles. Short development cycles, and the use of evolutionary development methodology, lead to the overlapping between development and maintenance. This overlapping is a typical aspect of the agile development methodologies.

The student who has the opportunity to work in a project in this way, is forced to work also for general refactoring and redesign activities, working for maintenance aspects for a good percentage of his time, becoming aware of the different problems and significance of this phase.

4.6 Regression tests

Another aspect hardly practicable during the study and directly related to the maintenance phase is the regression test ([9]).

Tests are accumulated during time, and their number grows proportionally to the application size. The most important value of tests, after their use during the development phase, is their regression aspect, i.e. the ability to inform the developer that everything that worked successfully before a given change or before a new release, still works correctly after the change has been effectively integrated into the application.

Precisely because related to the development time and to the application evolution, the importance of regression tests in a software engineering learning module is often only theory.

With the continuous integration platform that allows you to reduce the integration time and facilitate the development in short iterations, the role of regression tests is correctly emphasized.

4.7 Release management

The continuous integration platform, with the combination of a VCS system and an automatic build system, facilitates the management of different versions of a product.

The platform allows you to configure the version number of the build. In this way you can get unambiguous bundles (products), easily manageable and storable.

In the case of "branch", usually a maintenance version, which needs its own evolution, independent of the "trunk" (main version), you can manage the two elements as two different subprojects within the same platform, so that one does not influence the other, and, above all, so that you can work on one or the other, keeping track of the various versions delivered to the customer.

The release management is also an organizational aspect, rarely exercised in university courses.

4.8 Further values

Beyond the concept of continuous integration, introduced by agile development methodologies, but still less used and then to be considered innovative ([10]), especially in teaching and learning approaches, the use of an integration platform enhances and promotes distance work, development within distributed teams, and active participation on complex projects.

Projected into the real world, promoting distance activities does not only mean teaching and promoting outsourcing tools (still useful), but may also mean working from home (home office), with all what this implies in terms organization, quality of life, etc.

Learning to use such a tool gives added value to the e-learning practice itself. Indeed, you do not only use communication and learning tools from an academic point of view (the possibility to study at home), but you use them as real working tools (possibility of working from home), practicing them for your future working activity.

5. CONCLUSIONS

We explained and introduced the role of continuous integration in academic development projects, as mean to improve the development methodology on one side, and as mean to enrich the project-based learning experience of students on the other side.

We think that such a practice is so important that a software engineering lab activity should always be based on this. University bachelor or master courses that prepare students to actively enter the working world, should consider addressing this topic in a practical way, allowing the students to access such a platform and organize their software engineering projects.

The experimentation phase has yielded positive results. The students acquired more self-confidence in their development activities, easily monitoring their own work and the evolution of their projects. We are strongly convinced that this kind of experience will help our students to be proactive with newest development methodologies and practices in the companies where they will begin their working activities.

6. REFERENCES

[1] Apache Ant: http://ant.apache.org/.

[2] CVS: http://www.nongnu.org/cvs/.

[3] Eclipse: http://www.eclipse.org.

[4] JUnit: http://www.junit.org.

[5] Maven: http://maven.apache.org.

[6] Subversion: http://subversion.tigris.org/.

[7] TeamCity: http://www.jetbrains.com/teamcity.

[8] Duvall et al.: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, 336 pp., ISBN 978-0321336385, 2007.

[9] Massol Vincent, Husted Ted: JUnit in Action, Manning Publications Co., ISBN: 1930110995, 2003.

[10] Mastropietro R. et al.: Analysis and Evaluation of Cobra Information System, Final Report, internal report SUPSI-DTI, 2008.

[11] Moursund D.G.:Project-Based Learning Using Information Technology, International Society for Technology in Education: Eugene, OR, 2003.

[12] Pedrazzini Sandro: Tecniche di progettazione agile con Java: Design pattern, refactoring, test. Edizioni Tecniche Nuove, 298 pp., ISBN 88-481-1916-6, 2005 (Italian).

[13] Robert Cecil Martin: Agile Software Development. Principles, Patterns, and Practices. Prentice Hall International, 2002.

Remodelling Information Security Courses by Integrating Project-Based and Technology-Supported Education

Pino Caballero-Gil
Department of Statistics, O.R. and Computation
University of La Laguna
38271 Tenerife, Spain
pcaballe@ull.es

Jorge Ramió-Aguirre
School of Informatics
Polytechnic University of Madrid
28031 Madrid, Spain
jramio@eui.upm.es

ABSTRACT
The adoption of new teaching models based on student learning, as required by the European Higher Education Area (EHEA) should have deep implications both in curriculum and methodology. This paper proposes the use of Information Technology (IT) in a new teaching paradigm that fulfils EHEA objectives. It synthesizes previous experiences on teaching IT security and emphasizes both practical and theoretical aspects of teaching IT security. The proposal follows a project-based approach, which is very convenient in Computer Science (CS) education since it increases student motivation and provides application-oriented points of view. It describes two security courses for a CS degree, focused on Public Key Infrastructures (PKIs) and on wireless security. Both designed courses imply active learning in cooperative groups, what is useful in training of skills like leadership, communication and team work. The proposed approach exposes students to a successful result in terms of concepts understanding and motivation to learn.

Categories and Subject Descriptors
C.2.0 [**Computer-Communication Networks**]: Security and protection.

D.4.6 [**Operating Systems**]: Security and Protection.

E.3 [**Data Encryption]**

H.2.0 **[Database Management]**: Security, integrity, and protection.

K.4.4 [**Computers And Society**]: Electronic. Security.

K.6.5 [**Management Of Computing And Information Systems**]: Security and Protection.

General Terms
Design, Reliability, Security, Theory.

Keywords
Information Security, Curriculum Approach, Team Work, Computer Networks.

1. INTRODUCTION
The need of a certain standard of authenticity, confidentiality, integrity and availability of computer communications and data in our current technologic world has caused that IT security has become a subject of great interest because business nowadays cannot rely on the past concept of 'security through obscurity'.

Information security spending has increased much in the last years. The work [15] shows that while companies realize the importance of security, only 7% felt that they have the required skills and competencies to deal with existing and foreseeable security requirements effectively. Thus, a real need exists for competent and skilled IT security experts.

Teaching IT security is very complex as it includes diverse technical and non-technical contents and many theoretical and practical concepts. Universities need educators who can teach underlying theory to students in order to have them prepared to apply design principles to security mechanisms, but companies need immediate support in protecting their investments in people, equipment and data. In the last years, universities have introduced many subjects and courses to address IT security. This is a positive step, but many of them still implement traditional teaching approaches, and tend to forget that an important aspects of learning is to discover 'what', 'how' and 'why'.

This paper proposes an approach that gives equal emphasis on theoretical and practical teaching of information security subjects. It describes the approach through the description of two courses focused on PKIs and on wireless security subtopics as examples.

2. BASIC SECURITY DEFINITIONS
IT security is defined in [11] as 'the protection of information from unauthorized disclosure, modification, or loss of use by countering threats to that information arising from human or systems-generated activities, malicious or otherwise.' On the 'Federal Information Security Management Act' of NIST [7] 'information security' is defined as 'protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide: integrity, which means guarding against improper information modification or destruction, and includes ensuring information non repudiation and authenticity; confidentiality, which means preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information; availability, which means ensuring timely and reliable access to and use of information.' In other words, information security is there seen as a process both to protect integrity and confidentiality of information inside computer

systems and to guarantee availability of such information. It includes not only hardware, software and data in computer systems, but also management and personnel involved.

IT security must be understood as a broad concept involving computer, network and database security, cryptography, security management, computer forensics, and ethics among other concepts. Computer security includes the protection of the integrity of computer hardware and software. Network security is focussed on the security on computer network infrastructures. Database security protects mainly data integrity and data management. Cryptography provides the theoretical basis and software to guarantee mainly confidentiality, integrity, authenticity and privacy. Computer forensics identifies the computer laws and provides various tools to examine evidence of computer crimes. Security management controls the security process in practice. Computer ethics fundamentally educates computer users to avoid committing computer crimes. There are highly abstract mathematical concepts in cryptography, and philosophical views involved in computer ethics. The computer forensics must address not only the technical knowledge on computer systems and network environment, but also the laws about computer crime. Consequently, since IT security involves so many diverse knowledge and skills, teaching it effectively can be considered a challenging task.

3. EXISTING APPROACHES

Many papers describe implemented and evaluated security courses and resources [1,6,17]. In particular, the topics of IT security are often taught in a single course, but as [8] suggests, they should be taught separately.

According to Bishop [3] 'While security is discussed in some classes (such as operating systems courses), it has so many facets that only a course devoted to computer security can examine the basic underlying principles in detail.' He underlies a teaching IT security paradigm with three courses: a survey course, followed by two advanced courses on cryptography and system security.

IT security education has been also developed by Irvine et al. in [9], where two different approaches are proposed: 'Computer security could be the focus of the curriculum, which would investigate the foundations and technical approaches to security in considerable depth', and 'a computer science or computer engineering curriculum could choose to use computer security as an important property to be addressed in all coursework.'

The authors of [14] checked information security concepts from university curricula against knowledge defined in industry surveys. They classified 10 basic domains: 'security architectures and models, access control systems and methodologies, cryptography, network and telecommunications security, operating system security, program and application security, database security, business and management of information systems security, physical security and critical infrastructure protection and social, ethical, and legal considerations'.

4. THEORY VERSUS PRACTICE

The Computing Curricula project developed by the Association for Computing Machinery, the Association for Information Systems and the IEEE Computer Society in its version of 2005 [4] recommends positive weights of security topics across the five kinds of degree programs including security issues, principles, implementation and management. In particular, it describes these concepts as: 'Security: Issues and Principles - Theory and application of access control to computer systems and the information contained therein. Security: Implementation and Management - The organizational activities associated with the selection, procurement, implementation, configuration, and management of security processes and technologies for IT infrastructure and applications.' Consequently, according to the Computing Curricula both theory and applications are important in information security as the ability to abstract security concepts and apply formal reasoning techniques provides a fundamental basis and a broader perspective on security for students.

The authors of [12] state that the combination of theory and practice constitutes a tested way to ensure that students understand underlying principles, and think critically about issues such as limitations of computers and software correctness. It is also acknowledged in [2] that undergraduate students resist against the abstract nature of theoretical models and their relevance to real-world applications.

Many CS graduates will come in their future jobs with many promising secure solutions. Giving students a thorough understanding both of the theory and the practice that sustain IT security products maximizes their critical thinking ability to make informed decisions. Consequently, an important objective of CS teaching should include comprehension of a variety of IT security theoretical concepts, and knowledge about how such rigor is incorporated into computer systems and networks. On the other hand, it is often the practical and technical knowledge what distinguishes between a good and an excellent graduate. This work proposes a teaching approach with equal emphasis on theoretical and practical points of view of IT security.

5. PROPOSALS FOR TWO COURSES

Given that IT aspects of a 240 ECTS Computer Science degree are equivalent to about 480 teaching hours, a relevant question that deserves a thorough analysis is the amount of such time that should be devoted to IT security. A possible estimation can be derived from the demand for IT security education through its popularity in Masters Programmes.

We would suggest four compulsory security-specific courses: IT Security, Network Security, Cryptography and Security Management in order to provide students with all underlying security concepts. Unfortunately, as in other CS topics, undergraduate students do not always have the academic maturity to deal with some concepts. Thus, on the one hand, the necessary abstract mathematical concepts can be included in an advanced cryptography course, while on the other hand, the variety of concepts from the field of Security Management, can also be given in a CS master course.

Given the total available teaching time of 2400 hours for a CS degree, 5% corresponds to 120 hours (equivalently 12 ECTS), which represent a quarter of IT teaching time. Where security issues are offered, they are usually delivered in two 6 ECTS courses. Examples of two possible 6 ECTS information security courses are given in Tables 1 and 2. Both courses have been designed following a project-based and technology-supported

approach, which. is reinforced with lecture and readings from well-known and classical security literature and texts [10,13].

In the first proposed course (see Table 1) a project-based approach is carried out through the development of a Certification Authority (CA) and a PKI based on the concepts, protocols and tools introduced within the course. A possible option to carry out the project is through an open source implementation such as OpenSSL, which provides a functional cryptographic processing toolkit with a rudimentary CA, which requires much work to be developed as a fully functional CA. When the necessary concepts have been introduced, the project can be started and carried out by teams according to the following basic steps:

1. Set up of the CA software.
2. Generation of the CA certificate.
3. Generation of Certificate Signing Request (CSR).
4. Signature of CSR to generate signed certificates.

In the second proposed course on Wireless Security (see Table 2), the project-based approach requires students to be divided into an offensive team and a defensive team [16]. The main goal for the offensive team is to compromise wireless networks managed by the defensive team. Meanwhile, the defensive team task is to make sure that their wireless networks are secure from any type of attacks launched by the offensive teams. Attacks can range from exploiting vulnerabilities of standards and protocols, to the use of simple social engineering techniques, which will allow students to use their communication skills in order to try to obtain useful information like usernames and passwords.

Table 1. Outline of IT Security Course Contents Divided Into Weeks

Preliminaries and basic concepts	1. History of cryptography, basic information and number theory. 2. Basic security definitions. 3. Secret-Key Cryptography: stream and block ciphers, key management, random numbers 4. Public-Key Cryptography: exponentiation ciphers. 5. Digital signatures and hash functions: standards and applications.
Secure Communications	6. OpenPGP 7. S/MIME 8. S-HTTP 9. OpenSSL 10. IPSec
Access Control and Authentication	11. Fixed and one-time passwords. 12. Challenge-response schemes: Kerberos, smart cards.
Public Key Infrastructure	13. Certification and registration authorities. 14. Certificates: types, confidence guarantee, certification hierarchy. 15. Functions: certification and revocation. X.509 standard

Table 2. Outline of Wireless Security Course Contents Divided Into Weeks

Preliminaries and basic concepts	1. Introduction to wireless networking: configuration, detection, etc. 2. Introduction to wireless security issues: principles, requirements, etc. 3. Overview of threats and attacks concepts: risks, vulnerabilities, intrusion detection, malicious software, tests and audits, safeguards, and intrusion handling challenges, threats and hacking methodologies, message interception and modification, eavesdropping, denial of service, etc. 4. Intrusion tools & techniques: vulnerabilities analysis, packet capture & injection programs, sniffers, key crackers, traffic monitors, address spoofers, packets decryptors. wireless networks auditors., etc.
Wi-Fi Security	5. Wi-Fi technologies and security mechanisms. 6. IEEE 802.11b standard: WEP, RC4 encryption, vulnerabilities, etc. 7. Wifi Protected Access: WPA, TKIP, vulnerabilities, etc. 8. IEEE 802.1x protocol and 4-Way handshake: TLS over EAP, vulnerabilities, etc. 9. IEEE 802.11i: WPA2, AES, CCMP, AES-CCMP,
Security in Bluetooth	10. Encryption in Bluetooth: E0 Algorithm. 11. Secret key management and node authentication in Bluetooth
Security in MANETs	12. Encryption in MANETs 13. Authentication in MANETs 14. Routing and Cooperation in MANETs 15. Special MANETs: Sensor networks, VANETs, etc

The proposed approach requires the following basic steps by the defensive team:

1. Set up of a simple wireless network.
2. Use of basic security measures: change of default AP password and channel, disablement of SSID broadcasting, enablement of WEP keys and station MAC filter.
3. Implementation of security settings including replacing WEP implementations with WPA2, and deployment and configuration of an authentication server.

Simultaneously the offensive team must carry out the steps:

1. Set up of laptops with relevant operating systems by the offensive teams.

2. Wireless network traffic analysis: types of traffic, number of nodes and access points, signal strength, etc.
3. Attack sessions using the latest tools they can find.

In both courses, according to the project-based approach, during the first classes the project description must be given to the students in order to increase their motivation to follow the lectures. At the end, students or teams must give a presentation regarding techniques used to carry out the project and a demo. Virtual learning offers a flexible tool useful to support the project completion. The potential of IT and the Internet are used to hold a strategy that has the learner as focus, what is one of the main objectives of the EHEA. This combination of project-based and virtual learning-supported approach gives more freedom to students who must search and obtain information while filtering the relevant information obtained in the lectures.

6. DISCUSSION AND RESULTS

First, it is remarkable the absence in the proposals of many concepts very common in the security field, such as firewalls and hardening for example, but we have decided to follow specific work lines that do not include them explicitly. However, these and other topics will be dealt with by the students during their autonomous work when preparing the projects, and the virtual learning environment will be very helpful to solve possible doubts regarding concepts not included explicitly in the lectures outlines.

The main idea behind the proposal is the assumption that, as stated in [5], the basic task of a university teacher is to create situations of which students cannot escape without having learned. This also should be done through reasonable steps that lead to an ambitious final mission. Among possible activities that are useful to achieve such a goal, we remark the following:

1. Define clear and specific learning objectives.
2. Detail what students must do inside and outside class.
3. Establish deliverables with deadlines.
4. Propose works to be done in teams.
5. Stimulate cooperative learning.
6. Create feedback mechanisms.

Evidence supports that a mixed theoretical/practical approach helps students to learn difficult concepts. In the traditional lecture-based approach, most students do not get to understand the relevance and implications of the theoretical basis of the topic, mainly because they are not motivated. In our proposal, motivation is mainly obtained through the project-based approach as students are actively involved in the entire process of the project development, beginning from the early setup stages, installation and configuration to the final step of project presentation. Soft skills including communication and persuasion abilities and leadership are obtained thanks to the proposed project-based work-in-teams approach. Through the proposal, students realise that every technology has weaknesses and vulnerabilities, and that it is up to them to be aware and take actions to use and update technologies accordingly based on situation and circumstances. Students of the second course might also carry out a wireless network survey of the university wireless network, within a specified set of strict guidelines so that no unintentional harm can be done. This might indeed help the university to strengthen its wireless network security implementation and consequently could be considered as a social service offered by the students to the university.

7. CONCLUSIONS

Certain consensus on the content of curricula for IT security already exists. However, a methodological redefinition is needed. This work proposes a project-based approach that emphasizes both practical skills and theoretical competency in IT security. For a particular implementation, it suggests the topics of PKI and wireless networks. Since this is a work in progress, many open questions exist such as the evaluation of empirical data from the offer of the proposed courses.

8. ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Science &I, and FEDER under Project TIN2008-02236/TSI.

9. REFERENCES

[1] ACM JERIC (2006) Special issues on resources for the computer security and information assurance curriculum 6.

[2] Almstrum, V. L., Dean, C. N., Goelman, D., Hilburn, T. B., Smith, J. (2001) Support for teaching formal methods. Working Group Reports ITiCSE, 71-88.

[3] Bishop, M. (1993) Teaching Computer Security. Ninth IFIP SEC, 43-52.

[4] Computing Curricula (2005) ACM, AIS & IEEE.

[5] Cowan, J. (1999) On Becoming an Innovative University Teacher. Higher Education. Springer 37(4)

[6] Elorriaga, J.A., Gutiérrez, J., Ibáñez , J., Usandizaga, I. (1999) A proposal for a computer security course. ACM SIGCSE Bulletin 31(2) 42-47.

[7] Federal Information Security Management Act (2002) Public Law 107–347.

[8] Gaudin, S. (2003) Teaching Employees New Security Tricks. www.esecurityplanet.com

[9] Irvine, C.E., Chin, S-K., Frinke, D. (1998) Integrating Security into the Curriculum, IEEE Computer, 25-30.

[10] Menezes, A., Van Oorschot, P., Vanstone, S. (1996) Handbook of Applied Cryptography. CRC Press.

[11] NIST (2000) csrc.nist.gov/publications/nistbul/html-archive/oct-00.html

[12] Palmer, T. V., Pleasant, J. C. (1995) Attitudes toward the teaching of formal methods of software development in the undergraduate computer science curriculum: a survey. SIGCSE Bull. 27, 3, 53-59.

[13] Schneier, B. (1994) Applied Cryptography. Wiley.

[14] Theoharidou M., Gritzalis D. (2007) Common Body of Knowledge for Information Security. IEEE Security & Privacy 5(2), 64-67.

[15] Tohmatsu, D. T. (2007) 2007 Global Security Survey, Deloitte Touche Tohmatsu.

[16] Yurcik W., Doss D. (2001) Different approaches in the teaching of Information Systems Security," Information Systems Education Conference.

[17] Wagner, P.J., Wudi, J.M. (2004) Designing and implementing a cyberwar laboratory exercise for a computer security course. ACM SIGCSE Bulletin 36(1) 402-406.

TRAKLA2

Ari Korhonen
Helsinki University of Technology
P.O. Box 5400
02015 TKK
archie@cs.hut.fi

Juha Helminen
Helsinki University of Technology
P.O. Box 5400
02015 TKK
juha.helminen@cs.hut.fi

Ville Karavirta
Helsinki University of Technology
P.O. Box 5400
02015 TKK
vkaravir@cs.hut.fi

Otto Seppälä
Helsinki University of Technology
P.O. Box 5400
02015 TKK
oseppala@cs.hut.fi

ABSTRACT

TRAKLA2 is an online practicing environment for data structures and algorithms. The system includes visual algorithm simulation exercises, which promote the understanding of the logic and behaviour of several basic data structures and algorithms. One of the key features of the system is that the exercises are graded automatically and feedback for the learner is provided immediately.

The system has been used by many institutes worldwide. In addition, several studies conducted with the system have revealed that the learning results are similar to those obtained in closed labs if the tasks are the same. Thus, automatic assessment of visual algorithm simulation exercises provides a meaningful way to reduce the workload of grading the exercises while still maintaining good learning results.

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI), Distance learning; K.3.2 [**Computer and Information Science Education**]: Computer science education—*data structures and algorithms*

Keywords

automatic assessment and feedback, visual algorithm simulation exercises, data structures and algorithms, trakla2

1. INTRODUCTION

Understanding the fundamental principles of data structures and algorithms is necessary for all programmers, and the ability to write efficient programs based on this knowledge is an important skill. Learning the concepts involved requires both theoretical knowledge and program comprehension skills. TRAKLA2 [11] provides a graphical environment for learners to practice and test their understanding of data structures and algorithms by solving short exercises.

Visualizations are a key part of explaining how data structures and algorithms work. They provide a way to abstract out implementation details and leave what is essential in order to understand the state of a data structure. Drawing such visualizations using pen and paper is an often used element of both homework and examinations.

In a typical pen-and-paper exercise the student might be asked to draw images of an AVL tree when a given series of operations is performed. The main drawback of this type of an exercise is the amount of time and effort spent on drawing. In many cases the changes to data structures are also very localized, thus most of the time is spent copying from the previous image.

TRAKLA2 transforms these pen-and-paper exercises into electronic form. The underlying idea of simulating the algorithm is the same. However, instead of being forced to repeatedly draw the structures, the user is given a set of corresponding visualizations that can be manipulated with a mouse. The idea is to perform the same transformations made by the algorithm being simulated. This allows the student to concentrate better on studying the algorithm.

Having more time on task is not the only benefit. Visual algorithm simulation exercises can be automatically graded and feedback given when the learner is still engaged with the problem. The grading is based on comparing the learner-made simulation sequence to a sequence produced by an actual algorithm implementation. The feedback for the learner is based on the number of correct steps in this simulation sequence. TRAKLA2 also randomizes the input data for algorithms, thus it discourages cheating and allows more opportunities for practicing.

The system is still evolving. Currently we have some 50 ready-made exercises for data structures and algorithms covering most of the topics studied on a typical CS2 course. In addition, we have a set of advanced exercises, for example, to cover courses such as spatial data algorithms. In their current form the exercises can effectively be used to complement or replace existing simulation-based homework. In addition, we are in the process of including new areas of

Koli Calling '09 October 29-November 1, 2009, Koli, Finland

interest and a variety of new kind of exercises for basic programming courses.

The rest of the paper is divided into the following sections. Section 2 describes the system, and in Section 3 we discuss related tools. Section 4 explains how the system has been adopted and disseminated. Section 5 briefly introduces some of the numerous evaluations done on the system. Finally, Section 6 gives some future directions.

2. DESCRIPTION OF THE TOOL

TRAKLA2 is an environment for supporting the learning of data structures and algorithms. It is based on the Matrix algorithm simulation framework [5] written in the Java programming language. The system provides automatically assessed visual algorithm simulation exercises that are meant to be accompanied by some other study material such as lectures, lecturer's notes, textbook, etc. Indeed, TRAKLA2 is primarily intended to be used as a practicing environment rather than a stand-alone learning environment. The current selection of exercises includes operations on binary trees, heaps, sorting algorithms, various dictionaries, hashing methods, and graph algorithms. See Table 1 for a complete list of the simulation exercises.

The algorithm simulation exercises are solved in the following manner. The student simulates the operations carried out in a given algorithm by manipulating data structure visualizations. No coding or typing of text is required, since all manipulation is carried out in terms of graphical user interface operations. The system records the sequence of operations, and finally allows the student to submit it to the server. On the server side, the student's sequence is compared to a sequence generated by a working implementation of the algorithm, and the student is given immediate feedback. There exists also a version that does not need any server connection in which case the grades are not stored anywhere. Thus, the client can also check the solution, which reduces the time it takes to give feedback. The grade of the student's solution is stored on the server, and the teacher can monitor students' points and other statistics. After receiving the feedback, the student can retry the exercise, if needed. The initial data for the exercise, i.e. the input for the algorithm, is randomly generated for each try.

The primary user interface for the exercises (see Figure 1) is a Java applet. The applet provides visualizations of data structures, push buttons for requesting Reset, Submit, and Model solution for the exercise as well as buttons for browsing one's own solution backwards and forwards. Simulation is carried out by drag-and-dropping data items (e.g., keys to be sorted by a sorting algorithm or records to be inserted into a binary search tree) or references (i.e., pointers in linked lists and trees) from one position to another. Some of the exercises also include push buttons to perform exercise-specific operations such as rotations in trees.

TRAKLA2 has been released as open source under the Apache License. The Matrix framework required by the simulation exercises has been released under the GNU General Public License (GPL).

3. RELATED TOOLS

At least two systems provide exercises similar to the ones in TRAKLA2, namely MA&DA [8] and PILOT [1]. Neither of the tools, however, enable an instructor to manage student accounts and points. Furthermore, the sets of available exercises are not even nearly as comprehensive as in TRAKLA2.

In addition, there are numerous systems that visualize algorithms and programs, and provide other kinds of interaction ranging from algorithm animations to pop-up questions. However, the number of such tools is too extensive to be listed here, thus we only give a couple of examples. JHAVÉ [12] and ViLLE [3] both include pop-up questions in which the student is expected to predict some aspect of the animation. JHAVÉ is targeted to algorithm visualizations and ViLLE for program visualizations. Both systems have recently been integrated with TRAKLA2, however. Moreover, JHAVÉ and the corresponding TRAKLA2 exercises have been embedded into tutorials that deal with the algorithms in question, thus providing additional learning material. The differentiating characteristic of TRAKLA2 exercises is the aim to activate the learner to trace an algorithm (learning by doing) instead of merely observing the visualizations. The most important feature in this learning process is the feedback received from the system, which is automatically generated. The feedback is immediate and speeds up the assessment cycle, which we believe promotes learning.

4. DISSEMINATION

The hosting of the system has previously been provided by the Helsinki University of Technology with funding from the Ministry of Education in Finland for the Network project on basic programming studies to promote best practices. It has been employed by nine (9) different institutions in Finland: Helsinki University of Technology, University of Turku, Tampere University of Technology, Helsinki Polytechnic - Stadia, Lappeenranta University of Technology, University of Helsinki, University of Kuopio, Åbo Akademi University, and one in the US: Indiana University East, Richmond, Indiana. In total, the system has been used on over 50 courses, by over 6500 students who have submitted over 380,000 solutions to the exercises.

The system has been installed by other universities as well. In addition, there is also a commercial provider that hosts TRAKLA2 internationally (in Europe, Asia, Africa and Australia) as part of its service[1].

5. EVALUATIONS OF THE TOOL

Over the years, many evaluation studies about TRAKLA2 and the simulation exercises have been carried out. Here, we present the most important results and introduce the various research questions addressed. For a more complete list of studies, see the list of publications on the TRAKLA2 research site.

In one of the first studies, we compared the learning results of students solving algorithm simulation exercises in instructed classroom sessions with students using a web-based learning environment [7]. The study concluded that there was no significant difference in the final exam results between the randomized student groups if the exercises were the same. Thus, some of the classroom activities can be replaced with this type of web-based material.

Several studies have focused on the use of resubmissions in the system. One of the key findings is that an encouraging grading policy combined with the option to resubmit

[1] `http://www.bythemark.com/`

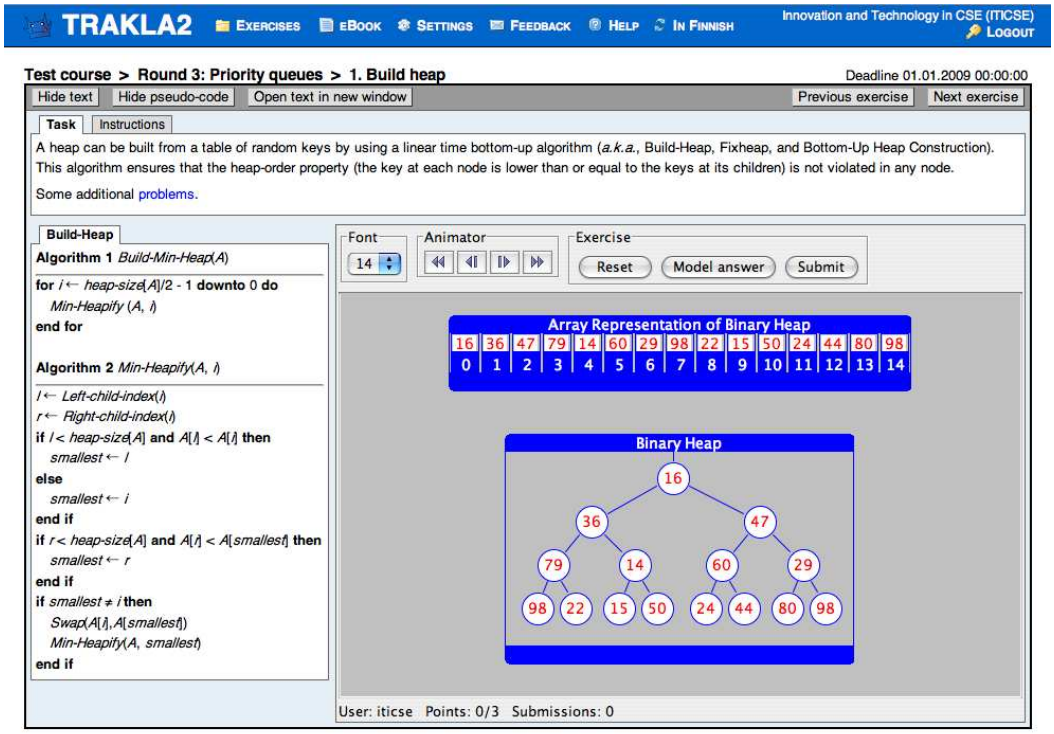


Figure 1: TRAKLA2 exercise view. The assignment instructions are located in the top area of the window together with some navigation buttons. The algorithm to be simulated is defined on the left and the data structures to be manipulated are on the right.

Table 1: List of available TRAKLA2 exercises.

Topic / Exercise	**Topic** / Exercise	**Topic** / Exercise
Basic algorithms	**Search trees**	**Algorithm Analysis**
Binary search	Binary Search Tree Search	Order of Growth
Interpolation search	Binary Search Tree Insertion	Running time of recursive algorithms
Preorder	Binary Search Tree Deletion	Running time of iterative algorithms
Preorder with stack	Faulty binary search tree	Asymptotic analysis
Inorder	Digital Search Tree	**Spatial Data Algorithms**
Postorder	Radix Search Tree	Point in Polygon
Levelorder	Single rotation	Point in Polygon with R-Tree
Postfix evaluation	Double rotation	Douglas-Peucker Line Simplification
Infix to Postfix	AVL-tree insertion	Closest pair of points
Dynamic programming	Red Black Tree	Point-Region Quadtree Insert
Sorting algorithms	Red Black Tree Coloring	R-Tree Insert
Quicksort	B-Tree	Polygon Traversal
Radix-exchange-sort	Trie	Line Sweep
Counting methods for sorting	**Hashing**	Visibility with Rotational Sweep
Insertion sort	Linear probing	Expanding Wave-method
Mergesort (iterative/recursive)	Quadratic probing	Voronoi Construction
Heapsort	Double hashing	Adding a point to TIN
Selection sort	Separate chaining	Polygon Skeleton
Priority queues	Rehashing	
Build heap	**Graph algorithms**	
Heap operations	BFS	
MinHeap Insert	DFS	
MinHeap Delete	Prim's algorithm	
	Dijkstra's algorithm	
	Dijkstra's algorithm with heap	

the solutions is an important factor in promoting students' learning [10]. In addition, the students solving more exercises get better grades. However, the article notes that in order to prevent the aimless trial-and-error method of problem solving, the number of resubmissions allowed per assignment should be carefully controlled. Fortunately, another study found that only a small number of students actually resubmit without thinking in between submissions [4].

In a more recent empirical study [9], collaborative learning on different Extended Engagement Taxonomy levels was examined with learning materials related to the binary heap that used visualizations on the different levels [6]. Pre- and post-tests were used as the test instruments in the experiment. In the study, statistically significant differences were found in favor of the students on the higher level of engagement, where also the TRAKLA2 exercises fall, in the total and pair average of the post-test scores.

In another study, students' simulation sequences recorded in TRAKLA2 were analyzed to infer existing student misconceptions of the build heap algorithm [13]. The results of this study suggest that misconceptions about how specific algorithms operate might be automatically recognizable from the simulation sequences. Such information would have high value for educators both for preventing and adressing misconceptions.

6. FUTURE WORK AND WEBSITE

In addition to the simulation exercises, other types of exercises and learning tools can be integrated into the system. Already, ViLLE system [3] provides exercises for the basic programming course (CS1) that can be incorporated into TRAKLA2. As mentioned, JHAVÉ [12] system has also been integrated into TRAKLA2. In the future, we are planning to include other similar tools such as Jype [2] to cover even more topics and courses. The idea is to avoid the need to learn to use many different systems by the learners, and thus focus on learning the content of the courses instead.

The project can be followed on a website that is located at `http://svg.cs.hut.fi/TRAKLA2/`. You can easily create a test account or utilize the stand-alone exercise applets without logging in. In addition, Koli Calling Conference maintains a website at `http://cs.joensuu.fi/kolistelut/tools/` that contains a brief overview of the system, a short video about the system, this paper, and some additional material.

7. ACKNOWLEDGMENTS

We thank the numerous people that have been involved in designing, implementing, testing, developing, and evaluating TRAKLA2 and its earlier versions during the past 18 years for their invaluable contributions.

This work was supported by the Academy of Finland under grant number 210947 as well as Ministry of Education, Finland.

8. REFERENCES

[1] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 139–143. ACM Press, New York, 2000.

[2] J. Helminen. Jype – an education-oriented integrated program visualization, visual debugging, and programming exercise tool for python. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, March 2009.

[3] E. Kaila, T. Rajala, M.-J. Laakso, and T. Salakoski. Automatic assessment of program visualization exercises. In A. Pears and L. Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*. Uppsala University, 2008.

[4] V. Karavirta, A. Korhonen, and L. Malmi. On the use of resubmissions in automatic assessment systems. *Computer Science Education*, 16(3):229 – 240, September 2006.

[5] A. Korhonen. *Visual Algorithm Simulation*. Doctoral dissertation (tech rep. no. tko-a40/03), Helsinki University of Technology, 2003.

[6] A. Korhonen, M.-J. Laakso, and N. Myller. How does algorithm visualization affect collaboration? video analysis of engagement and discussions. In J. Filipe and J. Cordeiro, editors, *Proceesing of the 5th International Conference on Web Information Systems and Technologies*, pages 479–488, WEBIST 2009, 23-26 March, Lisboa, Portugal, 2009. INSTICC — Institute for Systems and Technologies of Information, Control and Communication.

[7] A. Korhonen, L. Malmi, P. Myllyselkä, and P. Scheinin. Does it make a difference if students exercise on the web or in the classroom? In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 121–124, Aarhus, Denmark, 2002. ACM Press, New York.

[8] M. Krebs, T. Lauer, T. Ottmann, and S. Trahasch. Student-built algorithm visualizations for assessment: flexible generation, feedback and grading. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 281–285, New York, NY, USA, 2005. ACM Press.

[9] M.-J. Laakso, N. Myller, and A. Korhonen. Comparing learning performance of students using algorithm visualizations collaboratively on different engagement levels. *Journal of Educational Technology & Society*, 12(2):267–282, 2009.

[10] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal of Educational Resources in Computing*, 5(3), September 2005.

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

[12] T. Naps. JHAVÉ: Supporting Algorithm Visualization. *Computer Graphics and Applications, IEEE*, 25(5):49–55, 2005.

[13] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student misconceptions – a case study of the build-heap algorithm. *Computer Science Education*, 16(3):241–255, September 2006.

Web Eden: support for computing as construction?

Meurig Beynon
Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
wmb@dcs.warwick.ac.uk

Richard Myers
RJM Solutions
Haverflatt, Burrells
Appleby CA16 6EG, UK
Richard@rjmsolutions.com

Antony Harfield
Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
ant@dcs.warwick.ac.uk

ABSTRACT

As Ben-Ari has observed, whatever the merits of adopting a constructivist pedagogical stance towards Computer Science education (CSE), it is impossible to reconcile the classical view of computer science with a constructivist epistemology. There are nonetheless good reasons for wishing to invoke a broader epistemological framework in connection with modern developments in computing practice. These include: the extent to which computing technologies must be studied in the broader engineering context; the greater prominence that the experiential and phenomenological aspects of interaction with computers have acquired; the aspiration (e.g. in agile methodologies) to construct computer artefacts as an integral part of gaining the domain knowledge required for complex software development. This paper proposes Empirical Modelling (EM) as a constructivist pedagogical approach that promises to address such broader issues in CSE within a constructivist epistemological framework. In the light of Ben-Ari's insights, this is possible only through adopting an alternative view of the nature of computing. The Web Eden interpreter is introduced as a suitable first prototype for an EM tool to support this vision for "computing as construction".

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**], K.3.2 [**Computer and Information Science Education**]: Computer Science Education, D.2.6 [**Programming Environments**]: Interactive environments.

General Terms

Design, Experimentation, Human Factors, Languages, Theory.

Keywords

Computer Science Education, educational technology, epistemology, constructivism, Empirical Modelling.

1. CONSTRUCTIVISM AND COMPUTING

1.1 Issues for Computer Science Education

The educational emphasis of classical computer science reflects the perception of the computer as a reliable, predictable device suitable for performing computation in the sense identified by Stein [2]: "Computation is a function from its inputs to its output. It is made up of a sequence of functional steps that produce – at its end – some result that is its goal." Teaching programming is at the core of the classical discipline. Learning to program involves using formal languages whose syntax and semantics is not negotiable. As Ben-Ari observes [1], whilst CSE that respects this tradition may benefit from a constructivist pedagogical stance, it cannot embrace a constructivist epistemology such as has been the focus of controversy in the philosophy of science (cf. Latour [3]).

Koli Calling '09, October 29 - November 1, 2009, Koli, Finland.

Modern computing nonetheless provokes questions that are not easily addressed by traditional computer science. For instance:

a. How should we place classical Computer Science in the broader engineering context? Applying computing technology in complex systems raises concerns traditionally associated with engineering. In asking "What can we expect of formal verification?", the distinguished software consultant Michael Jackson stresses the need to take fuller account of the engineering perspective in complex systems development. And whilst Ben-Ari remarks upon the affinity between CSE and engineering education [1], he identifies the extent to which his findings generalise to engineering education as an open question.

b. To what extent are experiential and phenomenological concerns within the scope of Computer Science? In many modern applications of computing technology, the requirements relate as much to experience as to abstract function. The concrete physical characteristics of the technology itself then play an essential role. When we consider the relative merits of different devices and interfaces for speed-texting, for instance, we are led to think of the computer as resembling an instrument, and to recognise the impact that acquired skills have upon effective performance.

c. *Can we interpret programming activity as a legitimate way of developing domain understanding?* Agile methodologies are prominent in current software development. In such approaches – contrary to traditional programming precepts – the conception of the software product and the domain understanding this presumes are apparently developed even as the product is being constructed. Interpreting software as embodying domain understanding rather than merely meeting a functional requirement raises challenging philosophical and ontological questions (cf. Loomes and Jones [4]). The Play-In approach to software development advocated by David Harel illustrates a process of software construction that resembles the negotiation of meaning in a constructivist idiom.

Such questions all relate to how far we can conceive interaction with computers as "computational" in the narrow sense of Stein [2]. Accepting that interaction with computers must be program-like in this sense makes it hard even to *formulate* these questions. This has motivated many critiques of classical computer science.

1.2 Broader Visions of Computer Science

Discrepancies between computing practice and classical computer science theory have been noted by many researchers and interpreted in many different ways. Writing in 1998, Ben-Ari [1] remarked that "the gap between the standard libraries (especially the GUI libraries) of a modern programming environment and the model of the computer is so great that motivating beginners has become a serious problem". For Ben-Ari, the GUI libraries are obstacles to the appreciation of the computer as an "accessible ontological reality" of which the student must develop a mental model. By contrast, Winograd and Flores [5:78] contend that "computers do not exist, in the sense of things that possess objective features and functions, outside of language" and argue for a reconceptualisation of computing beyond the "rationalistic" epistemological framework. Ridley [6] articulates the perplexing issues that surround database theory, where the relational model that was once viewed as the foundational cornerstone of the field is widely perceived as inadequate to account for modern practice.

The fact that Boden [7:1414] reviews the history of the concept of computation under the heading "Computation as a Moving Target" reflects the subtlety of the notion. Cantwell-Smith [8] highlights the inadequacy of traditional accounts of computation in modern computing practice, and highlights the fact that what is understood by the "semantics of computation" in theoretical computer science is not to be confused with "the [content relation] that holds between the computational process and the world outside it" (which Smith describes as "the semantics of the semantics of the process"). Stein [2] argues for the need to move from the classical interpretation of "computation as calculation" to "something one might call computation as interaction".

These diverse critiques of classical computer science indicate that there is considerable interest in broadening the scope of the science of computing to embrace issues that cannot be addressed by focusing solely on the classical theory of computation. Ben-Ari [1] offers cogent reasons for believing that computer science as narrowly interpreted as the study of program-like interactions with computers cannot be based on an epistemological framework that embraces a constructivist stance. But whilst the critiques by Winograd and Flores, Cantwell-Smith, and Stein offer helpful insight into what an alternative science and an alternative epistemological framework might be like, they are ill-developed in respect of principles and tools, especially when viewed alongside Turing's profound mathematically-based contribution to our understanding of algorithmic processes.

1.3 Empirical Modelling

The approach to computing to which the Web Eden tool to be introduced in the second section of the paper relates is that of *Empirical Modelling* (EM) [9]. EM is based upon an unconventional epistemological framework that is consonant with William James's radical empiricist philosophical stance [10]. James's conception of knowing is rooted in direct experience – his primary thesis is that relationships between experiences are themselves given in experience. This is the basis on which one experience (e.g. managing one's expenses) can serve as the content of another (e.g. manipulating a spreadsheet). Though such knowing is of its essence a personal matter, this is no obstacle to its potential classification as having an objective quality, if indeed one's own experience is experienced as cohering with that of another person experiencing the same situation (cf. the way in which a financial spreadsheet can represent public information about a company's finances). The nuances to which such a concept of knowing can be adapted are sufficient to admit the kind of realist conception of a computer that Ben-Ari endorses [1], subject to certain reasonable contextual assumptions. It makes good sense to view a computer in this way when considering it as a computational device in a narrow sense for instance, but is not so appropriate if the experience of the computer that is the subject of concern is the colour of the display, or the possibility of erratic operation due to hardware failure is taken into account.

The basic thesis of EM is that there are fundamental and generic principles to help to construct artefacts intended to be experienced as having a specific content. The key to this construction is to introduce counterparts in the artifact for the relevant observables of its referent, and define dependency relations – automatically maintained as in a spreadsheet – to reflect the way in which changes to sets of observables are linked in latent atomic changes of state. In EM, the role of such artefacts – known as *construals* – is to mediate the modeller's experiential understanding of a situation before this can be articulated in propositional terms. Developing such construals is conceptually prior to programming activity. Like spreadsheets, construals primarily represent a current state of affairs or situation rather than a process.

EM engages directly with questions *a*, *b* and *c* above.

Because of the fundamental role it gives to personal experience, it is clearly intimately linked with *b*. The way in which EM invokes experiential and phenomenological concerns is well-oriented to an engineering perspective. EM principles can be applied to making sense of situations from the perspectives of human agents with different perceptions and capabilities. By imaginative projection ("to what observables subject to which dependencies can a thermostat respond, and which can it change?"), EM can be applied to other kinds of agent. Building construals is an activity that then discloses viable physical and interpretative mechanisms that might be exploited in applications. In this way, it lays the foundation for many different potential functional uses.

Conventional programming activity and the concerns of classical computer science can be interpreted as a specialised form of interaction within the broader framework that EM affords. The emphasis classical CSE puts on abstraction and logic reflects the fact that the empirical activities associated with identifying the computer as "an accessible ontological reality" [1] are a matter of prior engineering to be taken for granted. In contrast, EM addresses contexts where the nature and robustness of the would-be computational mechanisms is yet to be established [9:#087]. Such a reconceptualisation of computing enables the blending of engineering and classical computer science outlooks sought in *a*.

The radical nature of this reconceptualisation is highlighted by the insights that EM brings to question *c*. James's epistemological stance maintains that all knowing is ultimately rooted in connections that can be experienced. In EM, building construals is about relating knowing to its experiential roots. Though EM can lead to the realisation of program-like behaviours, this realization takes the form of an enactment of pre-rehearsed interactions within a constructed concrete live environment, rather than the specification of an abstract computational process optimised to a specific pre-conceived functional objective. On this basis, EM is an activity that supports the development of domain

understanding, but not an activity that can be properly viewed as programming. And where conventional CSE principles and tools are concerned with situations and interpretations that have been reliably pre-established and with associated knowledge that can be expressed in propositional form, the emphasis in EM is upon principles and tools that support the experimental learning activities that must precede such an understanding [9:#098]. It is for this reason that the principal EM tool, the EDEN interpreter to be introduced in section 2, is of its essence a technology to support learning without reference to any specific domain.

1.4 EM in relation to other critiques

EM has many points of contact with the critiques cited above. In EM, the emphasis in interpreting interactions with computers is on "the semantics of the semantics" in the sense of Smith [8]. There is scope for the negotiation of meaning that is relevant in particular to the social processes that frame the protocols for computer use and the identification of patterns of interaction and interpretation with devices that can be deemed to be program-like. As in Stein's conception of computation-as-interaction [2], much importance is attached to maintaining models of the external current state to which the computing activity refers (cf. for instance Stein's discussion of her use of "bootstrapping directly from physical interaction" to equip a robot with a capacity to read maps [2:19]). The realisation of system-like behaviours through the rehearsal and orchestration of primitive interactions amongst agents is well-aligned with the computational metaphor of "a community of interacting entities" proposed by Stein [2:9].

The crucial difference between EM and the proposals associated with the critiques mentioned above is that the development of EM has been intimately connected with identifying principles and building tools to support their application. These principles are more discriminating in the kinds of analysis and application that they endorse. For instance, in keeping with Ben-Ari's realist view of the nature of the computer [1], they legitimise Winograd and Flores's contention that "[computers] are created in the conversations human beings engage in when they cope with and anticipate breakdown" only in particular contexts. They likewise echo Ben-Ari's reservations about the scope for bricolage in conventional programming by calling into question Turkle and Papert's claims – cited by Stein [2:16] to support her concept of computation-as-interaction – about the amenability of traditional programs to experimental development [11]. And, because they focus upon "the semantics of the semantics" of a computational process rather than its abstract denotational/operational semantics, they challenge the notion that the "new generation of software engineering and design tools" identified by Stein in [2:16] illustrates a decisive shift from the usual computational metaphor.

2. THE WEB EDEN ENVIRONMENT

The Web Eden environment [12] is an online environment for constructing interactive models using EM principles. It represents a radical new concept in technology-enhanced learning (TEL) that has been applied in particular to CSE [9:#107], but – as motivated above – can address any learning domain. By exploiting non-standard principles based on modelling dependency relationships for software construction, it introduces a new paradigm for open source development that blends with the learning experience. Because of its distinctive approach to software construction, Web Eden affords an unusually intimate blending of domain learning with model-building in the spirit of Latour's construction [3, 9:#100]. This gives unprecedented scope for exploiting the environment to support learning in many different idioms. We can use Web Eden to guide learners through traditional tutorial-like learning material. Web Eden also enables the learner to explore live dynamic artefacts (as opposed to static pages of learning material). If the learners become really advanced, they are able to build their own artefacts and associated learning activities. Web Eden can run as a stand-alone environment, or we can embed it inside a virtual learning environment such as Moodle [9:#106].

Web Eden, like a spreadsheet environment, features counterparts of meaningful variable quantities ("observables"), defined connections between these which express the ways in which changing the value of one observable directly affects the value of another ("dependencies") and specific instances of redefinition of observables, both manual and automated, that correspond to meaningful action on the part of different agents. The use of dependency is a common – if implicit – feature of much educational software (e.g. tools like Mathematica, The Geometer's Sketchpad, AgentSheets and Matlab, and learning artefacts such as Cabri Geometry and Logotron's Visual Fractions), and its merits are endorsed by the wide range of educational applications for spreadsheets [13]. The motivating idea that makes Web Eden distinctive is that these merits cannot be fully realised within a conventional conceptual framework for computing [9:#096]. In particular, dependency cannot be integrated into an educational tool based on orthodox software principles (such as Imagine Logo) without compromising its conceptual integrity [9:#104].

Conventional TEL software offers little support for integrating the roles of the teacher (a pedagogical expert who conceives and specifies the educational content, interfaces, learning outcomes and exercises), the learner (typically a naive computer user who interacts with the learning environment through a preconceived interface) and the developer (an expert programmer who implements the environment). The Web Eden environment is open for interaction in all three roles at all times [9:#080]. What is more, the interaction takes essentially the same form for teacher, learners and developers alike. Every change to the current state to the current environment, no matter how it is to be interpreted (for instance, whether it is a change to the specification of the environment, a step in the learning process, or a revision to the interface or the underlying program), can be expressed as a redefinition of observables in the model. All restrictions upon interaction and interpretation are then of their essence purely discretionary, according to the expertise and interests associated with each specific role. This does not preclude the specification of interfaces to constrain the ways in which particular agents can redefine observables where this is appropriate.

Web Eden is a web-enabled version of the EDEN interpreter [9:#106]. EDEN was web-enabled by Richard Myers in a prize-winning final year computer science project at the University of Warwick in 2007-8. It exploits state-of-the-art tools that make it possible for server and client machines to share the computational load in interpreting a model. It also overcomes the problems of efficiently interpreting many EDEN models concurrently by enabling distributed processing and load-balancing over many EDEN virtual machines. Many hundreds of models have been built using EDEN [14]. All such model-building has a strong ingredient of domain learning. Many models have an explicit

educational objective and the range of learning applications is broad. Web Eden inherits the qualities of EDEN as an educational technology (cf. the "Applications Area" hyperlink at [9]), creating a platform for the full realisation of the pedagogical advantages for which previous experience of EDEN has offered proof-of-concept, and helping to overcome the practical obstacles to wider dissemination and adoption. It addresses the portability issues encountered in downloading the interpreter and models, simplifies the integration of the EDEN engine with other applications through the use of web interfaces, and is designed to incorporate session-sharing features that obviate the need to set up networks for collaborative and distributed modes of interaction.

The most comprehensive practical introduction to Web Eden and the modelling principles on which it is based can be found in the workshops prepared in conjunction with *The Sudoku Experience* - an online activity for gifted and talented pupils organised by the University of Warwick in July 2008 [12]. In these workshops, novice learners are first acquainted with the basic concepts and techniques that are required for model-building. This involves introspecting about the kinds of observables and dependencies that are significant in solving a Sudoku puzzle. They are then shown how these can be related to other tasks, such as devising formulae to convert between different ways of indexing the squares of a Sudoku grid. Once the principles of model-building have been introduced, their application to Sudoku solution is illustrated with reference to a "colour Sudoku" extension and the automation of a technique that is first conceived and implemented as a 'manually executed' pattern of interaction. In the final workshop, the Web Eden environment is configured to allow collaborative concurrent solution of Sudoku puzzles.

Web Eden was also applied in an online database module in the Virtual Studies in Computer Science (ViSCoS) programme at Joensuu University, Finland in 2008-9. This involved integrating Web Eden with the Moodle environment [9:#106]. In the module, design flaws in the international standard RDB language SQL are exposed by contrasting and critiquing different strategies for implementing SQL over a pure relational algebra notation. This practical and interactive approach to highlighting abstract design issues exploits the scope for open-ended interaction that Web Eden affords, which encompasses the capacity for implementing additional notations within the Web Eden environment on-the-fly.

The Web Eden Sudoku model was re-used in a second-year undergraduate module in December 2008. The Alloy tool for formal specification was used to generate the five essentially different abstract mathematical groups of order 8. To make the structure of these groups more accessible, the 9-by-9 grid in the colour Sudoku model was adapted for displaying and manipulating the corresponding group tables [12]. Simple patterns of redefinition and renaming of elements served to acquaint students without specialist mathematical knowledge with the character of a mathematician's intuitive, rather than purely abstract and axiomatic, understanding of group structure.

Other illustrative examples of the use of Web Eden can be accessed via [12]. The environment has recently been further developed to support more sophisticated online use with personal and public project data. The fact that the essential interaction with online models is mediated entirely through definition of observables prepares the ground for several significant extensions. These include: comprehensive monitoring of interactions that enables intermediate states to be recorded and revisited as if "live"; novel possibilities for collaboration primarily mediated through interaction with artefacts rather than communication based on language; potential for graphical user interfaces for fabricating scripts from templates. And though we have gathered informal evidence in support of our claims [9:090], we recognize the need for more rigorous evaluations through empirical studies.

We envisage the deployment of Web Eden not as the release of a product that meets a clearly preconceived specification, but as initiating an ongoing organic process of continuing development associated with the progressive extension, refinement and adaptation of existing models and of the environment itself to better meet educational goals. Teachers, developers and learners will all participate in this process. A major concern in TEL has been that of standardisation. In 2002-4, the principles underlying Web Eden were effectively deployed at the BBC R&D Laboratories in resolving critical issues of cross-platform portability of digital content. This gives us confidence that, appropriately deployed, Web Eden can offer rich experiences customised to diverse learners and contexts. To achieve this goal, we aspire to bring together representatives from schools, universities and industry worldwide to establish an online "Centre for Constructivist Computing" to promote the creation of models, teaching and learning strategies, and extensions and refinements of the modelling tool through open source development.

3. REFERENCES

[1] M. Ben-Ari. Constructivism in computer science education. SIGCSE Bulletin, 30(1):257--261, 1998.

[2] L.A.Stein, Challenging the Computational Metaphor, Cybernetics and Systems 30(6), September 1999, 1-35.

[3] B.Latour, The Promises of Constructivism, In Ihde, D. (ed.) *Chasing Technoscience: Matrix of Materiality*, 2006, 27-46.

[4] M.J.Loomes and S.V.Jones, Requirements Engineering: A Perspective Through Theory Building, Proc. ICRE'98, 1998, 100-107.

[5] T.Winograd and F.Flores, *Understanding Computers and Cognition*, Addison-Wesley, 1987

[6] M.J.Ridley, Database Systems or Database Theory, Proc. LTSN-ICS TLAD Workshop, Coventry, UK, 2003.

[7] M.Boden, *Mind as Machine: A History of Cognitive Science*, Volume 2, Clarendon Press, Oxford, 2006.

[8] B.Cantwell-Smith, Two Lessons of Logic, *Comput. Intell.* **3**, 214-218, 1987.

[9] Empirical Modelling website and EM papers as indexed at http://www.dcs.warwick.ac.uk/modelling

[10] W.James, Essays in Radical Empiricism, Bison Books, 1996.

[11] S.Turkle and S.Papert. Epistemological Pluralism: Styles and Voices within the Computer Culture, Journal of Women in Culture and Society 16(1): 128-157, 1990.

[12] http://www.warwick.ac.uk/go/webeden

[13] J.E.Baker, S.J.Sugden, Spreadsheets in Education: The First 25 Years, Spreadsheets in Education, 2003.

[14] http://empublic.dcs.warwick.ac.uk/projects/

Understanding open learning processes in a robotics class

Ilkka Jormanainen
University of Joensuu
Departmenf of Computer Science
P.O. Box 111
FI-80101 Joensuu, FINLAND
ijorma@cs.joensuu.fi

Meurig Beynon
University of Warwick
Department of Computer Science
Coventry CV4 7AL, UK
wmb@dcs.warwick.ac.uk

Erkki Sutinen
University of Joensuu
Departmenf of Computer Science
P.O. Box 111
FI-80101 Joensuu, FINLAND
sutinen@cs.joensuu.fi

ABSTRACT
Robotics is a functional approach for learning basic concepts of computing. In this role, it has been successfully used in introductory classes of Computer Science and Information Technology from primary to higher education. In a typical scenario, learners design and program robots in small groups that comprise between 2 and 4 students. Although such activity is stimulating for learners, the teacher of the class may find it hard to follow each individual student's learning processes. Empirical Modelling gives the teacher a platform to monitor individual group processes by collecting data from the construction and programming of the robots and allowing the teacher to model the empirically observed process. Unlike most adaptive learning systems, the model and the modelling process are transparent and open to the teacher, and even the students are able to assess their own learning based on the derived models.

Categories and Subject Descriptors
K.3.3 [**Computers and education**]: Computer uses in education – *Computer-assisted instruction (CAI).*

General Terms
Design, Human Factors

Keywords
Educational robotics, student modelling, Empirical Modelling, intervention, agency.

1. INTRODUCTION
Educational robotics has become a recognised tool for teaching at different school levels from kindergarten to university. The diversity of disciplines to which educational robotics has been applied is wide, and educational robotics is also a recognised part of computer science curricula [6]. The usual work process with educational robotics is based on group oriented working methods and open-ended problem solving. This readily leads students to take different paths to solving their problems, and groups may progress differently within a cycle of planning, building, programming, and testing. A robotics classroom might have 30 to 40 students divided into groups of 2 to 4 students. The unpredictable problem solving strategies and multiple student groups quite often cause the teacher to face difficulties in identifying the appropriate points for intervention. We are addressing this problem by utilising a system based on a multi-agent architecture [4] to support a teacher's observation process in the classroom. The agents can observe, for example, the students' construction and programming processes, as well as the teamwork and dynamics within and between the groups.

In this paper, we present an application for supporting a teacher in an educational robotics class. Based on the concept of conflative learning environment [4], we have built an environment for modelling the learning processes. The modelling is done with Empirical Modelling (EM) tools that allow open and transparent modelling of the learning process. The EM environment encourages role conflation, where a teacher can adopt a software developer's tasks in his or her own work and build in this way a support environment to match the current learning situation. The application allows the teacher and learners to build a model of the learning and group processes in a gradual way, based on the empirical data collected from the empirical observations arising from the current classroom setting.

Compared with traditional intelligent tutoring systems (ITS), the conflative learning environment framework provides a novel approach for the teacher to adapt the rules which form a base for modelling the students and learning processes. Instead of having predefined and static sets of rules, the teacher can construct the required rules from scratch by making use of logical operations to combine the atomic observations produced by the agents. Furthermore, the teacher can define what data should be collected, and how the data should be reflected to the model.

This paper is organised as follows. We first compare our approach to the previous work in the fields of ITS and adaptive systems. We then briefly describe Empirical Modelling. In section 4 we describe the conflative learning environment and a prototype application that we have built for deployment in educational robotics classes. Finally, we conclude the paper and sketch directions for future work in section 5.

Koli Calling '09, October 29 – November 1, 2009, Koli, Finland.

2. BACKGROUND

Educational robotics has become a recognised tool in many disciplines and school levels, including computer science education. In CS curricula, educational robotics has been used to teach both the basics of robotics as such and other computing concepts. Examples of the integration of robotics into the CS curriculum include for example teaching the Java byte code with the Lego robotics [3] and teaching systems-level programming topics by using the Lego robotics as a target platform [5].

Monitoring student groups' activities in the educational robotics classroom is difficult. Traditional intelligent tutoring systems have been applied also in this context [2]. However, as these systems are traditional programs with predefined specifications, they only offer the teacher a set of predefined options for interaction. The application based on the conflative learning environment framework provides support for open-ended exploration while having as its starting point the empirical observations of student activity. It is possible to support this kind of teaching process with traditional programming techniques and languages. However, where there are unpredictable scenarios, a single initial specification of a program is not enough to cater for all needs; the teacher also needs to have some degree of control over the development of the learning environment.

The traditional division of the roles in ITS and educational technology development processes usually strictly separates the roles of developer, teacher, and learner from each other. Moreover, the tasks undertaken by these process participants usually follow each other in a cycle with predefined steps. Beynon and Roe [1] argue that constructionist computer-assisted learning approaches can be seen as unifying the roles of the student, the teacher, and the developer. Following this line of argument, and invoking the concept of conflative learning environment described in [4], we can compare how the student and learning modelling process within the conflative learning environment framework differs from that associated with traditional ITS tools (Table 1). The main difference is that, whereas traditional ITS applications use a theory-based approach for building the learning model, the conflative approach starts from the empirical observations arising from the current learning situation. Another important aspect is that the EM based approach allows role conflation, and the tools are easier to adapt to different contexts and application areas.

Table 1. Comparison between the conflative and traditional tutoring approaches

	Conflative (EM-based) approach	Traditional ITS approach
Modelling approach	Empirical	Theory-based
Learning model	Constructed	Given
Adaptation	Transparent	Black box
Roles in the learning community	Flexible	Fixed
Direction of modelling	Bottom-up	Top-down
Modifications to the tools	On demand in the actual learning situation	Through the software development process

In the robotics classroom, the open-ended nature of robot building and programming typically leads to students taking completely different approaches to the activity. Accordingly, the teacher might not be satisfied with the existing sets of agents and rules for them, so that the system needs to be modified. In the traditional educational technology development process, the software developer does this. The developer can also make major modifications to the environment – for example, adding new data representations to the environment to create alternative views to record the students' progress. However, traditional software development methods are not flexible enough to support the teaching process within modern learning environments, where students explore solutions to problems independently.

3. ABOUT EMPIRICAL MODELLING

Empirical Modelling (EM) is a collection of principles and tools developed by Beynon, Russ and their students at the University of Warwick, UK. EM can be used to construct computer-based models that are based on the modeller's empirical observations about the phenomenon that is the subject of the modelling process. The modelling is done in the *tkeden* environment with several different notations.

The EM model is constructed by defining *observables* and *dependencies* with the notations mentioned above. An observable is a "computational" entity (such as a line, window, string or list of scalar values) that represents an element of the modelling subject. A dependency is a relationship between two or more observables. A key feature of the EM approach is that, after an initial definition, the EM environment automatically keeps the model updated according to the dependencies. This is similar to spreadsheet applications where values of the cells are updated automatically according to formulas that might contain references to other cells. In the next section we present through an example how the EM can be applied in the conflative learning environment.

4. A CONFLATIVE LEARNING ENVIRONMENT

To support the teacher's working process in a learning environment, such as an educational robotics class, where unpredictable learning activities often take place, we have proposed a concept of *conflative learning environment* (CLE) [4]. By exploiting the EM principles described earlier, the CLE gives full freedom for the teacher to modify the environment and support system to match the current situation.

The CLE framework consists of two parts. First, a number of *agents* work in the background of the learning process collecting observations about students' activity. Second, the teacher has a *model* constructed with the EM tools that reflects the current situation in the classroom, and the model-building is an ongoing process that accompanies the learning activities themselves. Each agent in the system has a dedicated task to which it has been appointed during the modelling process. For example, an agent might observe the use of a button in the graphical user interface of the robots' programming environment. This agent sends a message to the teacher's modelling environment over the network. The message can take the form of an EM definition so that the message redefines parts of the model. Alternatively, the message can be a natural language string that will be presented to the teacher as text. All types of messages contain a timestamp, and the messages are recorded in a database for later use.

Koli Calling International Conference on Computing Education Research Koli National Park, Finland, November 2009

The general idea is that agents do not process data by themselves, but collect data and deliver the data to the teacher's model and database for further observation. Even so, two different levels of "intelligence" can be distinguished within the agent population. The simplest form of agent works as a data collector. For example, an agent can observe a button in the robot's programming environment and send a message to the teacher's classroom model when students press that particular button. A more sophisticated agent possesses limited computing capabilities that enable it to do simple reasoning. For example, an agent can observe the existence of keywords or certain structures in the students' program code.

The working process in the educational robotic class usually takes a cyclic form. It is thus crucial that the teacher's tools also support cyclic working methods where the teacher can redefine the tool as needed when unpredictable events occur in the classroom. According to Empirical Modelling principles, the model is built up gradually by making redefinitions. The current state of the model is at all times captured by the set of definitions that have been introduced to date. Redefinitions can originate from both human participants and the automated agents, and these definitions then affect the model according to the current dependencies. The CLE periodically includes new definitions produced by the agents, and in this way the agents can automatically update the model according to the current situation.

It is obvious that there are technical challenges in using the EM tools to construct a learning environment. However, the teacher does not have to have expertise comparable to that of a technical developer. The most important thing is that the teacher utilises his or her expertise in the learning domain, and that the teacher has a clear understanding of the observables that mediate the learning activity. To make the EM-based conflative learning environment more accessible for the teacher, we propose that the modelling of the learning process should be divided into two parts. The first part, *technical modelling*, consists of setting up the basic modules of the environment. This part of the modelling process can take place before and even between the robotics classes, when the model can be redefined to meet the new requirements. The second part, *pedagogical modelling*, is the process that takes place during the classes. In this part of the modelling process, the teacher defines contextually meaningful observables and visualisations for the data that the agents collect. It is possible that these observables are usable in context-specific settings, for instance, for a particular class, or dependent on the phase where the students are in their project (building, programming, or testing).

4.1 A prototype application

By following the principles of constructing the conflative learning environment described in the previous sections, we have built a prototype environment to support teachers' intervention in the robotics classroom. The environment has been built gradually by following the cyclic process of EM model-building. As a starting point for the model building process, we conducted two experiments in which we collected data and analysed students' activities with a simple EM model as described in [4]. Based on the results and technical lessons learned from these experiments, we have constructed a model that can be used as a starting point for building a contextualised observation environment for different kinds of robotics classroom settings (Figure 1).

It is crucial to note that the application is an example, and most likely does not fulfil all the requirements of a teacher working in an educational robotics class. This is due to the fact that each teacher may want to observe different issues from the classroom and the learning process. We have built the application in such a way as to give a good overall impression about the use of the conflative learning environment framework and the potential of the EM tools in this kind of model building process.

With the application, a teacher can observe the progress of the student groups through various modules with graphical user interfaces. The modules are updated automatically as the agents make new observations and deliver them to the EM modelling environment. Furthermore, the teacher can simulate the students' progress subsequently based on the data that the agents have automatically collected and stored in a database. This post-processing can be also done with rules different from those used in modelling in the real-time situation. In this way, the teacher can potentially learn new things about students' actions and progress.

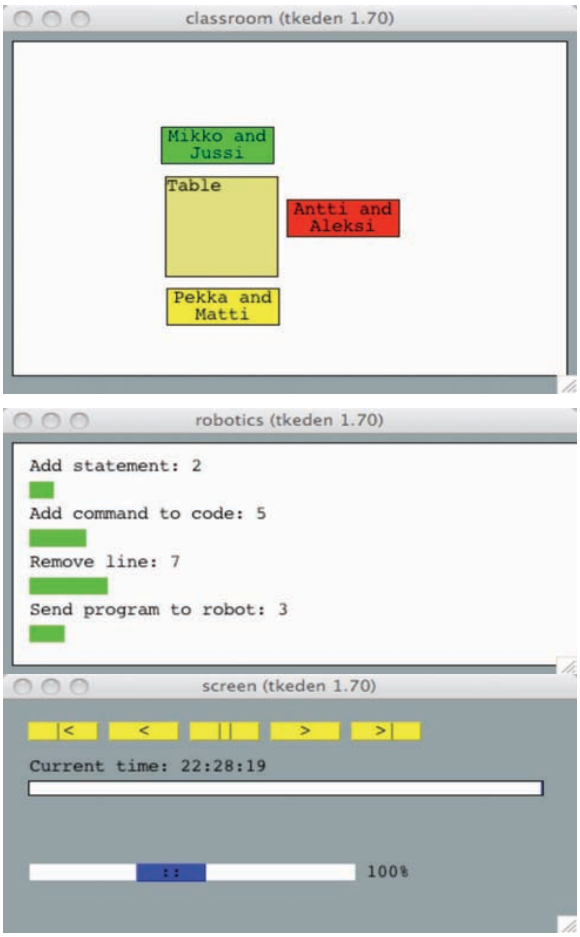


Figure 6. The prototype of the observation environment.

The current model consists of three modules. The first module (Figure 1, topmost window) shows a simplified map view for the classroom. The teacher can use this module to observe the overall progress of the student groups. In this prototype implementation, the student groups are shown as rectangles with the name of the group in it. These group markers can be moved

53

around on the screen to reflect the disposition of groups in the classroom. In the screenshot, one table has been modelled in the view and the student groups have been placed at the corresponding places around the table. The colour or size of the group marker can be bound by dependency to observables of interest – for example to the length of the program code that the student group has constructed so far. The model building is automated so that, besides automatically reflecting the agents' observations in the model, the groups are also appended or removed automatically from the model when they start or close the programming environment in the classroom. In this way, the model can readily be maintained to be consistent with the current situation in the learning setting. The second module (window in the middle, Figure 1) visualises the overall progress of the student groups as measured by a cumulative sum of clicks for the four most important buttons in the programming environment. The third module (lowermost window in Figure 1) implements replaying functionalities for the observation environment. By using the controls in the graphical user interface, the teacher can for example return to a certain moment in the learning process. All other modules are bound to this control module so that they will be updated to show the situation in the learning process in that particular moment of time. This module can also be used to process the data that has been automatically collected by the agents after the activity has finished, as opposed to in real time, and new rules and dependencies may be added for this purpose to give alternative views to the learning process. Reconstructing the live states of students' interactions so that the teacher can in principle experiment within these states in a fresh way is definitely one advantage of using the Empirical Modelling tools, and we argue that reconstructing the learning process like this is more difficult with traditional ITS tools.

This prototype application of three modules can be extended toward a more complete presentation of the classroom setting. As mentioned earlier, all visual elements can be redefined, and completely new views can be built to support the teacher as required in the current classroom situation.

4.2 Extending the application

An important aspect of the Empirical Modelling approach is the process of constant refinement of the model and the re-use of existing models. The EM repository[1] provides a catalogue of pre-existing models which can be modified to suit the new contexts. The adaptation of the existing models obviously requires a certain amount of work, and a technically oriented person should do this as part of the technical modelling process.

The conflative learning environment framework and the applications built on it can be also applied in other contexts. The data collection methods and learning process reconstruction tools are especially well-suited for deployment in other application areas. While building our robotics application, we applied the replaying module to an HIV/AIDS educational game. The new module allowed the teacher to replay students' actions in the game and analyse their thinking during the learning process. The adaptation of the existing module to a new context required very few changes to the original definitions, and the experience confirmed our view that Empirical Modelling can be used as an effective approach for constructing conflative learning environments.

[1] http://www.dcs.warwick.ac.uk/modelling

5. DISCUSSION

Recently, low-cost and highly accessible educational robot kits have gained popularity in hands-on learning environments, especially in technical fields, including Computer Science. However, the effective use of educational robotics in the classroom requires new kinds of classroom settings and teachers have to change their teaching methods according to the needs of the new environment. The open-ended nature of robot building can lead to students taking completely different approaches to an activity, and the teacher's needs for information about the learning process are difficult, if not impossible, to predict.

In this paper, we have presented a learning environment that allows the teacher to get and process information about the learning process through the empirical observations arising from the process itself. The application utilise the Empirical Modelling environment and model-building process to allow the teacher to modify the environment to meet the requirements of a particular learning process. Unlike most adaptive learning systems, the model and the modelling process are transparent and open to the teacher. The prototype application for monitoring robotics classes in its current form has been built through a cyclic process which took as its starting point empirical data collected from real classroom settings [4]. As the Empirical Modelling process characteristically involves a gradual open-ended development of the environment, we shall also develop the model further to provide the teacher better support in the classroom. In addition, we shall bring the modelling environment to students' screens, so that even the students are able to assess their own learning based on the derived models. This is a step towards a fully open and equal tutoring system where all participants in the learning community can participate in the modelling of the learning process by bringing to the model their own view of the activities.

6. REFERENCES

[1] Beynon, W. M. and Roe, C.P. 2004. Computer Support for Constructionism in Context. In Proceeding of the 4th IEEE International Conference on Advanced Learning Technologies, 216-220.

[2] George, S., and Despres, C. 1999. A multi-agent system for distance support in educational robotic. In the proceedings of the International Conference on Telecommunication for Education and Training, 344–353.

[3] Jipping, M. J., Calka, C., O'Neill, B., and Padilla, C. R. 2007. Teaching Students Java Bytecode Using Lego Mindstorms Robots. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education SIGCSE '07. ACM, New York, NY, 170-174.

[4] Jormanainen, I., Harfield, A., and Sutinen, E. 2009. Supporting Teacher Intervention in Unpredictable Learning Environments. In Proceedings of the 9th IEEE International Conference on Advanced Learning Technologies, 584 – 588.

[5] Klassner, F. and Continanza, C. 2007. Mindstorms Without Robotics: an Alternative to Simulations in Systems Courses. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education SIGCSE '07. ACM, New York, NY, 175-179.

[6] Sklar, E., Parsons, S., and Stone, P. 2004. Using RoboCup in University-Level Computer Science Education. Journal on Educational Resources in Computing 4 (2), 1–21

Mental Models of Data

Leigh Ann Sudol
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
lsudol@andrew.cmu.edu

Mark Stehlik
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
mjs@cs.cmu.edu

Sharon Carver
Psychology Department
Carnegie Mellon University
Pittsburgh, PA
sc0e@andrew.cmu.edu

ABSTRACT

This discussion paper describes the results of a pilot study into the mental models of data and data structures held by people based upon the software applications that they use frequently. Computers and data intensive software like email and iTunes have become a large part of our daily lives. The results are provided to motivate discussion about prevalent models of data and potential impact that they may have on introductory curriculum design. Results suggest that people tend to hold abstract models of the data types and regardless of gender or application being discussed there is a common structure applied by people.

Keywords

Mental Models, Introductory Programming, Computer Science Education

1. INTRODUCTION

Over the past 40 years introductory computer science education has favored making curriculum changes based on external demands such as popular programming languages or paradigms. This has introduced additional complexity into our courses, especially early on. In this time students have come to computer science class with increasing experience in using computers and computer software. The Common Sense Computing series of papers have sought to explore the naive concepts that students come to class with [8, 7].

As students have changed in their pre-existing knowledge over the last 40 years, so has the software that they work with on a daily basis. With an increase in social networking, communication and data storage and organization on the computer, the software applications in these domains have also become more sophisticated. Through repeated use of these applications, the current generation of students have developed sophisticated models of the information that drives the software as well as the interaction with this information.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

In this paper I explore the results of a pilot study aimed at assessing the way that people express the models that they have for working with this information. I discuss the methodology of data collection, the coding of the interviews, and the results. While not generalizable because of the small sample size, the results do point to a pre-existing model of data that is consistent across many applications and is based more in abstract representations and the interaction provided by the software application to the data. Finally, open questions prompted by this research are proposed for discussion.

2. MENTAL MODELS IN EDUCATION

The process of learning involves the connection between existing knowledge and new material. Without providing that connection, meaningful learning does not occur [5]. These two particular components of learning are equally important to be addressed, and yet often prior knowledge is not activated or assumed not to exist when instruction is going on. Many studies in cognitive science and conceptual change have shown us that attention must be paid to the knowledge and beliefs that students have prior to instruction[2].

A mental model is an internal representation of external objects or events[3]. They deal with both the form or structure of the item as well as the rules for interaction with the item[12]. Today's students have formed mental models for the data intensive applications that they work with on a daily or weekly basis. It is up to us as educators to take advantage of these models in our instruction.

With current instructional practices, students are not exiting introductory courses with appropriate models for computer science[4]. Without correctly identifying the models that students already hold before trying to impose new models, it is difficult to know if information is being lost in translation.

Cognitive science tells us that utilizing pre-existing models can help reduce cognitive load of students and help teachers find the zone of proximal development that is appropriate for their students[2, 10, 11]. Studies of conceptual change show that if students hold misconceptions of conflicting models with new information, their ability to apply and transfer that information correctly will be impaired[9]. While much of the research has addressed situations where the conflicting models are incorrect[1], the existance of competing models interferring in problem solving domains is well documented[6].

This pilot study asks the question of whether prevalent mental models exist for data intensive software applications,

and if so, can those models be aligned with data structures introduced in early computer science classes.

3. RESEARCH DESIGN

Information was gathered from subjects through in-depth interviews. There were 8 subjects interviewed from a private research university(7) and a public university(1), varying in computer science experience and gender. The categories of computer science experience were:

- No Programming:the person had never taken a programming class.
- Programming/Non Major:the person had taken 1 or more computer science classes but was not a computer science major.
- CS Major:The person was an upperclassman pursuing a computer science degree.
- Faculty: the person was a faculty member in either CS or Engineering at the university and had an extensive programming background.

Subjects answered questions about the data intensive programs that they used on a regular basis. Prior to being interviewed participants completed an intake survey that asked how often they used programs such as email, iTunes, mapping software (like Google Maps), a cell phone address book, instant messenger, Facebook, and You Tube.

Participants were asked the same four questions about each application they used frequently (more than once a week). The questions were designed to find out about both the structure and relevant operations on teh data held within the mental models. The questions were:

1. How do you normally interact with the program?
2. How do you keep your information organized? If you had to imagine how the computer saves or stores your information what would that look like?
3. If you want to get to one particular piece of information, how do you get there?
4. Imagine i want to change the way you interact with the software program so that you can see only one piece of data at a time. Which of the following four buttons do you think would be the most used? Get First, Get Last, Get Item Number , or Search? Which of the buttons do you think would be the easiest to make?

The questions were always asked in this order for all subjects and applications. As participants answered the questions, the interviewer may have asked for clarification if the answer was vague.

4. CODING

The data was coded at the utterance level from each subject. For the purposes of this analysis an utterance was defined as an idea that was expressed either in a statement or series of statements. A total of 228 utterances were recorded and coded. For each utterance, the application they were discussing, the question they were answering, and the time on the video was recorded. Each statement was coded as a reference to one or more data types based upon a table of structures and important behaviors of common data types. Data types included in the coding were Array, List, Map, Set, Matrix, Stack/Queue, and Graph.

For example the following student statement was coded as a Set because it referred to the unordered nature of the data in her perception. "Its like a big amorphous cloud and you as the user get to choose the way that you organize it." The utterance "I think its an actual list - most recently added, and from that list other lists are referenceable, like categories" was coded as both a list and also stack/queue. It made a reference to the word list (which implies a linear structure), as well as most recently added, which implies a time sequence.

The data was also coded based upon the interactions you could have with the information. A table of important behaviors for each data structure was created and used to code the utterances which dealt with data interactions. The java api was used as a reference for the kind of operations attached to each data structure. The concept of search, for example, was coded as both a List and a Set operation because both of those data structures contain a built in search feature. For example, the student utterance "Normally IŠm looking for a specific friend, I donŠt browse around too often, I just use the search bar" was coded as List and Set for this reason.

The table of structure vs. behavior is available at the author's web site in the appendix of this paper at

`http://www.cs.cmu.edu/~lsudol.`

5. RESULTS

Although the small subject size makes it hard to generalize, the consistency across very different subjects in a variety of domains offers strong suggestions of underlying structures.

5.1 Application Data

The 8 subjects were asked about applications they used more than once a week. There were 8 different applications that were covered, with an average of 4.6 applications per person. The four questions were asked about each application that participants indicated they used frequently on an intake survey. For each application the number of relevant utterances range from 2 to 12 per person.

5.2 Data Type Difference

There were several patterns that emerged across people and applications. The common language for List and Set included reference to the operation of searching, the ability to see if an item was contained within the data, or to look at all the items at once. Statements coded as stack/queue frequently involved the mention of a "first" or "last" item and the idea that order was preserved was very strong. Statements coded for arrays indicated rigidity (in terms of size or ordering) and indexes for each item.

5.3 Overall Utterance Count

As you can see in Figure 1; List and Set had the most utterances of all the data structures. The graph shows by data type the count of the utterance across all subjects and applications. Remember that one utterance may be coded in different categories.

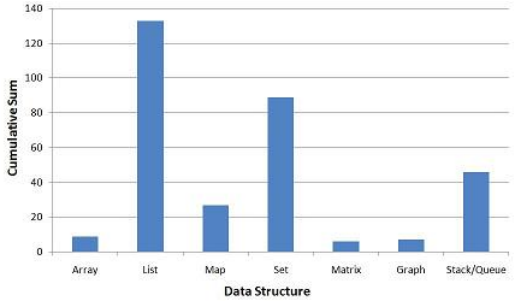


Figure 1: Utterance count by data type

5.4 Subject Background

The results can also be looked at by subject background. Presented here are the percentages since there were different numbers of utterances total per subject.

You will notice in the data from Table 1, that although arrays are often the first way that students are taught to store multiple records within a program, there is little evidence from this pilot that people have prevalent models for arrays. Even after taking a computer science course, people still tend to refer to abstract, dynamic models of data.

Table 1: Table 1:Data Structures by Subject Background

	No Prog.	Non Major	Major	Faculty
Array	3%	5%	2%	0%
List	38%	28%	34%	45%
Map	6%	5%	7%	17%
Set	33%	17%	26%	20%
Matrix	0%	3%	2%	2%
Graph	0%	3%	5%	1%
Stack/Queue	10%	20%	12%	9%

Although these results should not be considered generalizable, faculty members showed a strong bias towards List based data structures, while students who had never taken computer science were much higher for set or unordered structures. While these two results would be difficult to generalize to a larger population, it does motivate further study into what types of models people hold of data, and how that can be mapped to the type of data storage that students could encounter in an early programming course.

5.5 Gender Results

Table 2 shows the results by gender. While the numbers vary slightly the same overall pattern of in increase in the List and Set categories holds across both genders. While we often encounter discussion the gender differences between students studying computer science, there seems to be little evidence at this time for the idea that men and women perceive the information in the computer differently. However, in future studies gender will be recorded to validate this on a larger, more diverse population.

5.6 Results by Application

One question that needs to be explored is whether it was the applications discussed that prompted the pattern observed, or whether common models exist across all applications that get used on a regular basis. Table 3 shows the breakdown by utterance count across all of the applications.

Table 2: Table 2: Results by Gender

	Male	Female
Array	2%	4%
List	30%	36%
Map	9%	3%
Set	24%	27%
Matrix	2%	1%
Graph	2%	2%
Stack/Queue	15%	14%

Table 3: Table 3: Utterances by Application

	Array	List	Map	Set	Matrix	Graph	S/Q
Torent	1	12	2	4	0	0	3
CellPhone	1	21	6	12	0	0	7
Email	3	34	4	29	1	0	19
Facebook	1	23	6	15	1	2	9
Google Maps	2	15	1	8	4	4	2
IM	1	4	4	5	0	0	3
iTunes	0	17	3	14	0	0	2
YouTube	0	7	1	2	0	1	3
Totals	9	133	27	89	6	7	48

Despite both the differences in the visual and interactive aspects of the data contained throughout the different application, the pattern of responses remain the same. A majority of the utterances code as either List or Set based data structures. This consistency, despite the different software applications that were discussed implies a commonality to the models that the students hold.

6. OPEN QUESTIONS

This pilot study opens several questions for discussion.

- Does taking a computer science course (or many courses) change the way which we percieve data storage and interactions?
- Are there mental models that are prevalent in the student population before they take a computer science course?
- Should introductory education focus on utilizing preexisting models to sequence our instruction, or should we teach our standard curriculum because it is the standard curriculum?
- What other features of the mental models should be explored in future studies?
- If these results are replicated in a larger study, what implications does that have for computer science education and computer science education research?

7. ACKNOWLEDGMENTS

The research reported here was supported by the Institute of Education Sciences, US Department of Education, through Grant R305B040063 to Carnegie Mellon University.

The opinions expressed are those of the authors and do not represent the views of the Institute or the US Department of Education

8. REFERENCES

[1] M. T. H. Chi, J. D. Slotta, and N de Leeuw. From things to processes: A theory of conceptual change for learning science concepts. *Learning and Instruction*, 4:27–43, 1994.

[2] National Research Council. *How People Learn: Brain, Mind, Experience and School*. National Academy Press, 2000.

[3] Kenneth Craik. The comprehension of the everyday physical environment. *Journal of the American Planning Association*, 34:29–37, 1939.

[4] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmrs. *SIGCSE 2007: Proceedings of the 38th SIGCSE technical symposium on computer science education*, pages 499–503, 2007.

[5] Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, 1981.

[6] George J. Posner, Kenneth Strike, Peter Hweson, and William Gertzog. Accommodation of a scientific conception: Toward a theory of conceptual change. *Science Education*, 66:211–227, 1982.

[7] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Common sense computing(episode 4): Debugging. *Computer Science Education*, 18:117–133, 2008.

[8] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Commonsense computing: what students know before we teach (episode 1: sorting). In *ICER '06: Proceedings of the second international workshop on Computing education research*, pages 29–40, New York, NY, USA, 2006. ACM.

[9] S. Vosniadou. The cognitive-situative divide and the problem of conceptual change. *Educational Psychologist*, 42:55–66, 2007.

[10] Stella Vosniadou. Mental models in conceptual development. *Model Based Reasoning: Science, Technology, Values*, 1, 2002.

[11] L. S. Vygotsky. *Mind in Society: The development of higher psychological processes*. Harvard University Press, 1978.

[12] R. M. Young. *Mental Models*, chapter Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices, pages 32–52. Erlbaum, 1983.

Quick Introduction to Programming with an Integrated Code Editor, Automatic Assessment and Visual Debugging Tool – Work in Progress

Juha Helminen*, Lauri Malmi, Ari Korhonen
Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, 02015 TKK, Finland
{juha.helminen, lma, archie}@cs.hut.fi

ABSTRACT

Research into programming education has led to the development of a multitude of tools to support teaching and learning programming. The tools typically focus on a certain aspect of learning. Visualization tools support building conceptual level understanding of how programs work. Automatic assessment tools give feedback on submitted tasks. Specialized learning environments, such as microworlds restrict the number of concepts to be mastered or simplify writing programs by providing a limited set of operations and simplified syntax. In this paper we present a novel tool Jype that integrates a number of essential activities into one programming environment, and thus partially solves the problem of using many different tools on the first programming course. Design of Jype is based on knowledge that research has revealed about challenges in learning programming. Jype integrates program visualization features, visual debugging facilities including reverse execution, and automatic assessment on programming assignments. Moreover, Jype is designed for teaching Python, when most research in programming education support tools is based on Java.

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI), Distance learning; K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Human Factors

Keywords

computer science education, automatic assessment, visual debugging, jype

*Corresponding author.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

1. INTRODUCTION

Programming is a core competence in computer science (CS) and is usually the first step in a CS curriculum. However, students cannot program at the expected level of competency after having completed their introductory programming courses. This has been shown time and again in multinational and multi-institutional studies [15, 21, 23]. A commonly suggested explanation for the weak performance is that they lack adequate skills in problem-solving [19]. However, a significant source of the difficulties actually appears to be students' overall *fragile knowledge* of elementary programming and insufficient understanding of control flow and program state [13]. Fragile knowledge means that while one might possess the knowledge to answer direct questions about a particular subject they still might not be able to apply that knowledge in solving a problem on their own. In the programming context, this means that while students might be able to answer correctly to isolated questions on specific programming items they still are not able to combine this knowledge in order to write and understand the execution of a complete functional program.

The introductory programming courses are often tightly paced and students are expected to learn a new way of thinking about problems and a set of skills for working with programs in a relatively short period of time. Indeed, students are easily discouraged at the start of their first programming course as they are faced with the overwhelming complexity of programming tools combined with learning the many abstract concepts and notations. Not surprisingly, high dropout rates are common on first programming courses. In a recent study on the reasons behind this problem among CS minors at their institution, Kinnunen and Malmi [9] reported a rate of 26 percent, and in general the rate at many institutions is estimated to be at 20–40 percent. In the study, the students viewed finding run-time errors as the most difficult programming-related issue. They even went as far as to name the difficulty of tracking down even simple errors as one of the reasons behind their decisions to drop out [8]. As programming dominates the beginning of any CS curriculum, this difficulty may even discourage learners from continuing in this field.

At the heart of programming is the concept of *program comprehension*, which refers to the process of understanding programs and software. In essence, a programmer constructs

a *mental model*, an internal representation of their understanding about a program's intent, its data and execution, and this cognitive process is the focus of program comprehension research [22]. Indeed, what beginning programmers above all must develop, is a conceptual understanding of the computer's execution model, the *notional machine*, that is implied by the programming language's constructs. Consequently, many kinds of software tools have been built to aid in developing this understanding. Most of these incorporate some form of visualization in an attempt to better communicate the abstract concepts. Ultimately, however, learning to program simply requires practice. Learners must first gain an understanding of the basic mechanics and then practice logical reasoning by combining and applying their mental models to solve problems in a variety of contexts. In other words, a programming course must have many programming assignments to drill the students and let them properly evolve *viable* mental models. The models are viable if they allow the learner to accurately and consistently explain the mechanics of the constructs.

The practical goal of an introductory programming course is to learn to read and write programs. As discussed, in addition to adequate practice, the key to learning to program is having an accurate understanding of what constitutes a program's state and how a program is executed. Like with any abstract constructs, we can use illustrations to try to convey information about them. Thus, a widely-used method of supporting the learning process is to provide a visual representation of program state. There exist many program visualization tools and integrated development environments (IDE) aimed towards beginners that employ software visualization. Jeliot 3 [16] and jGRASP [1] are state-of-the-art examples of these. When students are given the ability to visually explore programs and algorithms we expect them to be able to better make sense of program executions and programming concepts. However, there is some controversy and conflicting research on whether visualizations by itself, such as animations of program executions, actually improve learning. Studies indicate that the level of interactivity is a significant factor in the learning outcome [4, 6, 17]. As opposed to passive animations, more engaging visualizations that activate the students appear to be more beneficial. This result has inspired researchers to create systems that integrate software visualization and *automatic assessment*: students are given tasks related to a visualization and their answers are automatically evaluated for correctness to give them immediate feedback. The assignments can, for example, be pop-up questions based on a code animation [11]. A typical question would ask the student to predict the value of a variable after the current statement has been executed.

Automatic assessment of coding exercises makes it possible to keep the teachers' workload manageable even on large courses, while still providing students with a reasonable level of guidance in developing their skills through hands-on experience with practical programming tasks. The systems can be broadly categorized into two classes. On the one hand, there are many systems that primarily provide a server for submitting solutions to get automatic feedback and provide no means of devising the solutions in the system itself and thus require the use of a separate editor. On the other hand, there are systems that, in an attempt to lower the barrier-to-entry, provide an integrated facility to write the code for the solution. In these systems the focus is on elementary programming. Some examples of the first category are BOSS [7] and Web-CAT [3] and the second JavaBat [18], WebTasks [20] and Javala [12]. None, however, provide more than a mere text editor for writing the solutions and they must be debugged elsewhere. VIP [24] is the one tool that closely integrates C++ programming exercises, in the sense of writing code to solve automatically assessed programming problems, with visualizations of program state so as to provide a built-in facility for visually debugging the solutions. However, our introductory programming course is built around Python. Overall, not many supporting tools have yet been developed for teaching Python programming. There are many professional IDEs and some microworld programming environments, such as rur-ple[1], Guido van Robot[2] and Turtlet[3]. ViLLE [11] also provides some support for visualizing Python execution with functionality for translating a Java code animation to an equivalent animation in Python.

2. DESIGN AND IMPLEMENTATION

In light of the problems presented, we designed a new education-oriented tool [5] for our newly established introductory programming course built around Python. We set five primary goals for our tool.

1. Facilitate program comprehension and aid in forming an accurate mental model of program state and execution through consistent automatically generated visualizations.
2. Aid in tracking down the causes of programming errors with integrated visual source code level debugging functionality that supports backstepping.
3. Engage students with automatically assessed programming assignments to enable and support the learning of programming, in the sense of actually writing code, by practice and repetition.
4. In achieving goals 1-3, add as little overhead as possible to the actual process of writing program code.
5. Minimize the barrier to entry by implementing the system as an easy-to-use web application, which also allows it to fully support independent distance learning.

Goals 1 and 3 are a response to the students' apparent fragile knowledge of elementary programming. The students are given meaningful coding tasks, which they are expected to solve by utilizing the integrated visual debugging functionality that provides a viable representation of the notional machine and is thus intended to guide them towards program comprehension. Goal 2 tries to address the problem of locating errors in a program, which students can easily get stuck on. The purpose of Goals 4 and 5 is to let students gradually build confidence in programming in a streamlined environment before moving on to more professional tools with steeper learning curves. That is, the intent is to soften the start of the course and give students a sense of what is fun about programming with a simplified easy-to-use interface, integrated functionality and an immediate initial focus on accomplishing actual programming tasks. We think this will provide students with a sense of achievement early on and then better motivate them to put the effort into learning to use the programming tools, and thus even possibly

[1]http://sourceforge.net/projects/rur-ple/
[2]http://gvr.sourceforge.net/
[3]http://www2.lut.fi/jukasuri/Kilppari/

reduce the number of dropouts.

The tool, known as *Jype*, provides functionality for visual debugging with integrated coding exercises that are automatically assessed. Figure 1 shows a screenshot of Jype.

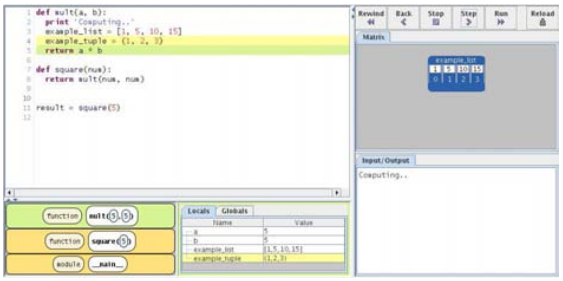


Figure 1: The basic view of Jype.

The visualizations are designed to facilitate the understanding of basic data flow and procedural control flow. A program can be executed step-by-step in a typical source level debugger line-by-line manner. Additionally, the execution can be stepped backwards. Program state is visualized at each step. The progress of control flow is shown by highlighting the line that is currently being executed and the line that was previously executed. Additionally, Jype provides a representation of the execution stack to illustrate scoping and the chain of function calls and returns. Program data is shown with a table-like view where changes are highlighted.

In contrast to a traditional debugger, the easy controls for stepping through the execution without inserting breakpoints allow quick and easy transitions between writing and visually tracing code. There are no compilation or build steps. Furthermore, execution can be stepped in reverse. Because the automatic feedback received from the automatic assessment is integrated to the coding environment, the student can fluidly move on to visually debugging the solution. Our hypothesis is that the easy controls and the integration of these functionalities would lower the threshold for visually tracing the program to find out problems. This way, students should locate errors more efficiently compared to more ad hoc approaches, such as adding print-commands and many forms of trial-and-error strategies.

Jype is implemented as a Java application that can be run as an Applet, as a Java Web Start application or as a local stand-alone application. It is built on top of existing open source libraries Jython[4], Matrix [10] and jEdit[5]. Jython is a Java implementation of Python used to run the Python code in Jype. Matrix is a data structure (DS) visualization library which in Jype is used to provide abstract visualizations of some DSs and to store visualizations as an animation sequence to allow for backstepping. Finally, Jype includes the code editor from the jEdit open source project.

The automatic assessment in Jype is implemented by using the standard Python unit test library. Tests, scoring and feedback is defined in terms of tests that exercise the student's program. Jype also includes a testing framework for defining output-based assessment where the output of the student's program is compared to that of a model solution with the same input. Jype provides only the environment for solving programming exercises and it is meant to be used with a course management system that is used to store and manage submissions and points. Currently, Jype exercises can be deployed on a TRAKLA2 [14] server.

[4]http://www.jython.org/
[5]http://www.jedit.org/

3. DISCUSSION

Jype is currently being used on a course for the first time and there are yet no results on the students' experiences with it. The question that immediately comes to mind with the development of a tool such as Jype is **how much there is a real need for this type of specialized tool for an easy introduction to programming on university level CS1 courses?** That is, should we—instead of scaffolding—simply provide more guidance and step into utilizing a real programming environment right from the start? This would most probably require to put more effort into teaching how to use these tools correctly and efficiently in order to cushion the start of programming. However, they must learn to use these tools at some point of their programming curriculum anyway. On the other hand, while extremely simple, Jype is yet another tool to learn and "grow out off", which leads to the idea of combining these two. The scaffolding could be integrated into a professional tool. For example, the functionality of Jype could be built into an Eclipse perspective. This could perhaps allow a more fluid transition into the real programming environment, but still necessitates setting up a programming environment as opposed to simply starting up a web application. The bottom line is, should the focus of future efforts in programming education tools be on extending and modifying existing professional environments or to continue creating one-off tools?

There is also the question whether or not this kind of integration of assessment with the programming environment is a good goal to strive for? **How is it going to to affect students' working strategies?** We would like to think that tight integration of easy-to-use visual debugging features with the assignments lowers the threshold of visually tracking errors expressed in the feedback. Thus, it would promote more successful debugging strategies. However, this may very well not be true. We have seen in other systems—allowing automatic assessment of programming assignments—that students might become reliant upon the scaffolding and never learn to program on their own. Without any restrictions on the use of the automatic assessment, the tendency could be to edit the code in a trial-and-error manner based on the automatic feedback until the tests pass. Simple solution is to limit the number of allowed submissions. Still, some students might more easily resort to trial-and-error strategies if they can get too immediate feedback on the correctness of their solutions. The bottom line is, how to diminish scaffolding during the course in order to push students towards evaluating the correctness of their solution more carefully before submitting it?

Finally, a point worth of discussion is whether it is the students' lack of adequate practice in elementary programming that is the underlying reason of their weak performance. In other words, **is it all about a need for more repetition on the basics?** That is, instead of having one or two programming assignments on branching and loops, why not have 3 or 6? Of course, having many simple programming exercises is laborious both for the instructor who designs them and for the student who has to solve them. However,

this is one of the problems that Jype and similar web-based systems try to address. By providing a streamlined environment that does not require the students to create a file for every few lines of code, and to upload the file to the assessment system, we can allow more repetition with short exercises. In her doctoral thesis [2], Eckerdal discusses programming assignments that incorporate too many different concepts simultaneously. Instead, to properly build an understanding of the concepts, they should first be singled out in individual exercises, and there should be enough variations of the same idea to solidify the understanding. The bottom line is, can we achieve this by developing learning environments that can deliver exercises with enough variation and repetition for students to master programming?

4. REFERENCES

[1] J. Cross, T. Hendrix, and L. Barowski. Integrating Multiple Approaches for Interacting with Dynamic Data Structure Visualizations. *Proceedings of the 5th Program Visualization Workshop*, pages 3–10, 2008.

[2] A. Eckerdal. *Novice Programming Students' Learning of Concepts and Practise*. Doctoral dissertation, Department of Information technology, Uppsala University, Sweden, 2009.

[3] S. Edwards. Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications*, volume 3, 2003.

[4] S. Grissom, M. McNally, and T. Naps. Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. *Proceedings of the 2003 ACM symposium on Software visualization*, pages 87–94, 2003.

[5] J. Helminen. Jype – an education-oriented integrated program visualization, visual debugging, and programming exercise tool for python. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, March 2009.

[6] C. Hundhausen, S. Douglas, and J. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[7] M. Joy, N. Griffiths, and R. Boyatt. The BOSS Online Submission and Assessment System. *Journal on Educational Resources in Computing*, 5(3), 2005.

[8] P. Kinnunen and L. Malmi. Why students drop out CS1 course? *Proceedings of the 2006 International Workshop on Computing Education Research*, pages 97–108, 2006.

[9] P. Kinnunen and L. Malmi. CS Minors' CS1 course? *Proceedings of the 2008 International Workshop on Computing Education Research*, 2008.

[10] A. Korhonen, L. Malmi, P. Silvasti, V. Karavirta, J. Lönnberg, J. Nikander, K. Stålnacke, and P. Ihantola. Matrix – a framework for interactive software visualization. Research Report TKO-B 154/04, Laboratory of Information Processing Science, Department of Computer Science and Engineering, Helsinki University of Technology, Finland, 2004.

[11] M. Laakso, E. Kaila, T. Rajala, and T. Salakoski. Define and Visualize Your First Programming Language. In *8th IEEE International Conference on Advanced Learning Technologies*, pages 324–326, 2008.

[12] T. Lehtonen. Javala – Addictive E-Learning of the Java Programming Language. In *Proceedings of The 5th Koli Calling Conference on Computer Science Education*, pages 41–48, November 2005.

[13] R. Lister, O. Seppälä, B. Simon, L. Thomas, E. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, and R. McCartney. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin*, 36(4):119–150, 2004.

[14] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[15] M. McCracken, T. Wilusz, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. Kolikant, C. Laxer, L. Thomas, and I. Utting. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[16] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing Programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, 2004.

[17] T. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, and S. Rodger. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bulletin*, 35(2):131–152, 2003.

[18] N. Parlante. Nifty reflections. *SIGCSE Bulletin*, 39(2):25–26, 2007.

[19] A. Robins, J. Rountree, and N. Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[20] G. Rößling and S. Hartte. Webtasks: Online programming exercises made easy. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 363–363, 2008.

[21] B. Simon, R. Lister, and S. Fincher. Multi-Institutional Computer Science Education Research: A Review of Recent Studies of Novice Understanding. *Frontiers in Education Conference, 36th Annual*, pages 12–17, 2006.

[22] M. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.

[23] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, and R. McCartney. Students Designing Software: A Multi-National, Multi-Institutional Study. *Informatics in Education*, 4(1):143–162, 2005.

[24] A. Virtanen, E. Lahtinen, and H. Järvinen. VIP, A Visual Interpreter for Learning Introductory Programming with C++. *Proceedings of The 5th Koli Calling Conference on Computer Science Education*, pages 125–130, 2005.

Diagnostic Web-based Monitoring in CS1

Olle Bälter
KTH CSC
100 44 STOCKHOLM
SWEDEN
+46 8 790 6341
balter@kth.se

ABSTRACT
Students that fall behind during a course are a concern in any teaching situation. Falling behind has negative effects both for students, teachers and the university. Close monitoring of the learning and development can be effective, but is in general time-consuming and expensive. The use of a web-based diagnostic system that can generate a large (infinite) number of questions could make monitoring both time and cost effective.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, learning.*

General Terms
Experimentation.

Keywords
Computer Science Education, Pedagogy, Generic questions.

1. INTRODUCTION
Science teachers too often experience how a student approach them towards the end of a course and reveal that they did not understand the topic of the 2nd week of the course and therefore have been unable to understand the rest. Teachers are of course aware of this problem and therefore introduce various minor tests and/or lab assignments before the final to promote continuous learning. However, as s a natural part of laboratory assignments there are also lot of support available from teachers and assistants. While this support is essential to help some students forward it can also be unintentionally misleading for some that can produce lab results (reports or in computer science: source code), but without understanding exactly why.

In some cases teachers blame the students that do not study or seek help early enough, but after spending a semester at an American top college with excellent students and still observing the same phenomena, it is clear that this happens even among very talented students. In general, an experienced teacher get a sense rather quickly which students are in danger of failing, but without hard evidence of the case it is difficult to initiate a discussion with the student. The teacher may be wrong, and the student may be in denial.

If we take the idea of assessment during the course to an extreme, we would constantly be assessing the students. This might have benefits, but takes time from teaching and interaction with the students and also feels a lot like baby-sitting.

One alternative that adds only a little workload to the teacher is to ask the students to hand in reflections over their learning. However, although beneficial in many ways, it adds to the workload for the students, and students with authoring skills may hand in reflections that seems right, but still has misunderstood some concepts, in similar ways that a verbally skilled student may slip through an oral examination of a lab assignment.

The solution should therefore minimize the time spent both for the teacher and the students and contain precise questions that can be assessed automatically. This way, the teacher only have to read a summary of the results and does not have to spend any time reading answers that are correct, which normally should be the vast majority.

2. RELATED RESEARCH
Introductory courses in computer science is a constant topic of discussion among academics and the hurdles to learn programming have been lowered by various tools, such as narratives, visual programming, robots, Lego [15] and visualizations of programs [14, 16]. One of the criticisms is that many students do not know how to program after an introductory course [13] and that programming assignments are subject to plagiarism [5]. Students report that it is acceptable to copy the majority of an assignment from a friend [19] and in one study, 40% of students plagiarized at least one assignment [3]. There are reports of 20% of the students failing the course [12].

Computer Assisted Assessment (CAA) is often presented as THE solution for education of the future. The opinions on Computer Aided Assessment (CAA) is split among academics, but there are claims that this is mostly due to experience with CAA[3]. There are several systems for CAA [11, 12] and there are studies that report no significant difference in examination between online exercising and classroom exercising [8]. Among the advantages with CAA is the possibility to personalize assignments and to resubmit answers (which is important from a constructivist perspective), which improved grades greatly, but the share of failing students remain approximately the same (slightly under 20% in [12]).

Koli Calling'09, October 29–November 1, 2009, Koli, Finland.

However, there are also disadvantages with CAA [24]. If CAA is used on the web, the problem of knowing who is answering the questions and also whether this person is receiving help or not [3] becomes difficult. One negative aspects of CAA not mentioned in the literature (maybe because it is too obvious) is that constructing problems and evaluations that are correct becomes even more essential, as slips, mistakes and errors cannot be handled as smoothly as on a written exam or a lab assignment where that teacher simply can admit the mistake and correct it immediately.

In order to improve learning, counter plagiarism and reduce the number of failing students (regardless whether they fail the course or not), we also need to improve assessment.

Lecturers often do not know how well students are doing until after the first assessment. At this point, it may be too late to prevent struggling students from falling [1]. We therefore need to assess students early and with problems suitable for their learning. We know that deep approach to learning not surprisingly leads to higher grades [20], but also that students' expectation of their own grade on the introductory course is the most important indicator of performance [18] and the students' comfort level is the best predictor of success [21], and the strongest relationship between fifteen factors and performance on a programming module was a student's perception of their understanding of the module [1]. One study suggest that weaker students should only be required to gain the ability to read and understand programs, and thereby demonstrating knowledge and comprehension (using Bloom's taxonomy) [9]. The initial assessment on these levels should be a part of the early identification of struggling students. Passing these simpler problems could strengthen their self-confidence and perception of the subject and thereby improve learning in the entire course.

There is at least one report of weekly tests [23], but this was made in labs, which of course reduce time for interaction. However, these weekly quizzes dramatically reduced failure rates [23] and lab exams are better assessors of programming ability than traditional methods such as written exams and programming assignments [5] but an examination of novice programmers and the SOLO (Structure of the Observed Learning Outcome) taxonomy ends with a recommendation to mix training and assessment of reading and writing tasks [10].

There are already online programming assessment tools [11, 17], but unlike the proposal in [17], we are only suggesting a pedagogical methodology, not a technical system. There is also an argument against combined development and assessment systems: the students are not learning to use the tools used in "real" development. Self-assessment has also been used successfully for terminology quizzes as a way to encourage reading lecture material before class [22], our proposal goes a little further.

3. PROPOSAL

A web-based system for small diagnostic tests would liberate students from coordinating assessments in time and place and the teacher could automatically be sent a summary by email. However, creating sufficient number of questions in such a system would be a very time-consuming endeavor, and if the number of questions is too few, there is always a risk that some students will copy answers from others.

A remedy to this problem is to use *generic* questions. A generic question is a question formulated in a way that makes it possible to construct a large (even infinite) number of questions from it. For example, as a first problem in CS1, the following code is provided:

```
a = 17

b = a

a = 42
```

and the question follows: what is the value of b? Depending on whether a and b are primitive or reference variables, the answer will be 17 or 42, respectively. Examining the question we can realize that a can be replaced with any valid variable name, as can b; and 17 and 42 can be replaced with any variable value.

Similar constructions can easily be transferred to mathematics (and are in use in web courses at our university) and possible to other science subjects as well. The foundation is that the question is *determinable* (that is, all answers can be classified as 100% right or 100% wrong) and *input dependent* (that is, there is input to the question and this input can be varied and effect output).

The web technology makes it possible to give students a small test every day (or before or after a lecture, a lab etc.). A student that fail the test may, thanks to the generic formula, be given a new test immediately in the spirit of constructivism (this idea was proposed by Keller in 1968 [6, 7]). This monitoring could improve the situation for students, teachers and the university.

From a student perspective this system could improve

- learning, as the tests will inspire some students to study first,
- clarifying whether the student has understood or not (it is easy to think you can because everything seems so simple when the teacher explains),
- teacher support, as failing the test repeatedly will give a clear signal that the student needs assistance, both to the student and the teacher.

From a teacher perspective the system can give information in several ways:

- Individual level: which students failed (more than once on a question) this can be used to approach these students to give them support
- Group level: Reports on how many (percentage) have failed (the first time) on each question and use that to repeat instructions during the course and improve the explanation to the next course
- With test results stored in a database it could also be used to detect negative trends (students that never use to fail suddenly fails)

From a university perspective the system could improve:

- Throughput of students as failures can be detected and corrected much earlier.
- Results in general as study habits improve.
- If the system is used in several courses, it could also be used to identify students that struggle in several subjects (many failures in several courses).

We have experience of similar attempts from the test system in a previous project [2]. A minor part of that project is still in use in

```
Introduction to Computer Science
Summary of test 1 September 9 2009

General
85% passed on their first attempt
10% passed on their second attempt
5% needed more than two attempts

Students with none or more than two attempts
Adam Sandler: 7 Failed
Britney Spears: 3 Passed

Questions
Attempts: 1  2  3  3+
Q1        45  3  2
Q2        48  2
Q3        45  3  1  1
```

Figure 1. Example of output from the system to the

on-campus courses in programming for the mid-term. The main difference between this project and previous is

- We now have an infrastructure for development and maintenance of the system.
- The focus on generic questions that has undisputedly right or wrong answers.
- The technical solution is far more evolved with a complete database, logging of web activities, etc.

Initially we will start with courses in computer science where we have most knowledge and experience, but we clearly see how these ideas can be extended into other sciences and for parts of social science and humanities.

4. STUDY DESIGN

There is a web-based system in use at our university, but today it is only used for distance education. This system will be a stable foundation for the experiments we intend to perform. There are generic questions on math and programming in the system, but these are ad hoc and there is no general way to introduce new questions, and no teacher interface for this.

In order to add functionality for generic questions and student monitoring we intend to:

- Observe how the ad-hoc solution to generic exam questions that are in use today can be used for diagnostic purposes, as proposed.
- Based on these observations, propose a design for the system so that any teacher can add new generic questions and use the system.
- Develop a model for organization and continuous improvement of a database with generic questions that all teachers have access to.

We will do this in two simultaneous pilot studies, one at an American college, and one at a Swedish university. At both sites the pilot will be run in an introductory course in computer science using Java or Python.

An example of output from the system to the teacher can be found in Figure 1. This could be sent via email to the teacher or be shown on a secure web page. The amount of information presented should be limited and the thresholds should be possible to configure. The information is divided in three sections. The general section is to get a sense for how the entire class has done on the test. The student section lists names of students with more than two attempts. The purpose is that the teacher should get information on which students that may need extra support. In this mock example, Britney Spears does not seem to need any assistance but Adam Sandler definitely does. The section with questions is not interesting at all in this example, but in case there is something wrong with one of the questions it will be clear which from this information. This may be attributed to a failure in formulating the question or if the question is correct, there might be something overseen in the teaching and course material.

5. DISCUSSION QUESTIONS

- Is this a good idea from a teacher's perspective?
- Is this a good idea from a student's perspective?
- How should the pilots be evaluated?
- Are there more efficient ways to achieve the same goals?

6. ACKNOWLEDGMENTS

Thanks to the project Virtual Campus at Resource Centre for Net-based Education, KTH Royal Institute of Technology for financing and STINT, Swedish Foundation for International Cooperation in Research and Higher Education for additional travel grants.

7. REFERENCES

[1] Bergin, S. and Reilly, R. 2005. Programming: factors that influence success. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA, February 23 - 27, 2005). SIGCSE '05. ACM, New York, NY, 411-415.

[2] Bälter, O. 2004. WIKKED (in Swedish) URL: www.nada.kth.se/utbildning/projekt/wikked Last Visited June 8, 2009.

[3] Carter, J., Ala-Mutka, K., Fuller, U., Dick, M., English, J., Fone, W., and Sheard, J. 2003. How shall we assess this?. In *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education* (Thessaloniki, Greece, June 30 - July 02, 2003). D. Finkel, Ed. ITiCSE-WGR '03. ACM, New York, NY, 107-123.

[4] Daly, C. and Horgan, J.M., (2001), Automatic Plagiarism Detection, Proceedings of the IASTED International

Conference Applied Informatics, pp.255-259. Innsbruck, Austria, Feb 2001.

[5] Daly, C. and Waldron, J. 2004. Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA, March 03 - 07, 2004). SIGCSE '04. ACM, New York, NY, 210-213.

[6] Herzberg P. 2001. The Keller Plan: 25 Years of Personal Experience. In Positive Pedagogy – Successful and Innivative Strategies in Higher Education, vol. 1, #1. ISSN: 1496-8126.

[7] Keller F S. 1968. "Goodbye, teacher..." J. Appl. Behavioral Analysis. Vol 1(1), pp 79-89.

[8] Korhonen, A., Malmi, L., Myllyselkä, P., and Scheinin, P. 2002. Does it make a difference if students exercise on the web or in the classroom?. In *Proceedings of the 7th Annual Conference on innovation and Technology in Computer Science Education* (Aarhus, Denmark, June 24 - 28, 2002). ITiCSE '02. ACM, New York, NY, 121-124.

[9] Lister, R. and Leaney, J. 2003. Introductory programming, criterion-referencing, and bloom. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA, February 19 - 23, 2003). SIGCSE '03. ACM, New York, NY, 143-147.

[10] Lister, R., Simon, Thompson, E., Whalley, J. L., and Prasad, C. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Bologna, Italy, June 26 - 28, 2006). ITICSE '06. ACM, New York, NY, 118-122.

[11] Malmi, L., Karavirta, V., Korhonen, A. and Nikander, J. (2005): Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. In ACM Journal of Educational Resources in Computing, 5 (3)

[12] Malmi, L., Korhonen, A., and Saikkonen, R. 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of the 7th Annual Conference on innovation and Technology in Computer Science Education* (Aarhus, Denmark, June 24 - 28, 2002). ITiCSE '02. ACM, New York, NY, 55-59.

[13] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education* (Canterbury, UK). ITiCSE-WGR '01. ACM, New York, NY, 125-180.

[14] Naps, T. L., Eagan, J. R., and Norton, L. L. 2000. JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations. In Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education (Austin, Texas, United States, March 07 - 12, 2000). S. Haller, Ed. SIGCSE '00. ACM, New York, NY, 109-113. DOI= http://doi.acm.org/10.1145/330908.331829

[15] Powers, K., Gross, P., Cooper, S., McNally, M., Goldman, K. J., Proulx, V., and Carlisle, M. 2006. Tools for teaching introductory programming: what works? In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA, March 03 - 05, 2006). SIGCSE '06. ACM, New York, NY, 560-561.

[16] Rajala, T., Laakso, M.-J., Kaila, E. and Salakoski, T. (2007). VILLE - a language-independent program visualization tool. In Proc. Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Koli National Park, Finland. CRPIT, 88. Lister, R. and Simon, Eds. ACS. 151-159.

[17] Roberts, G. H. and Verbyla, J. L. 2003. An online programming assessment tool. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia). T. Greening and R. Lister, Eds. Conferences in Research and Practice in Information Technology Series, vol. 140. Australian Computer Society, Darlinghurst, Australia, 69-75.

[18] Rountree, N., Rountree, J., and Robins, A. 2002. Predictors of success and failure in a CS1 course. *SIGCSE Bull.* 34, 4 (Dec. 2002), 121-124.

[19] Sheard, J., Dick, M., Markham, S., Macdonald, I., and Walsh, M. 2002. Cheating and plagiarism: perceptions and practices of first year IT students. In *Proceedings of the 7th Annual Conference on innovation and Technology in Computer Science Education* (Aarhus, Denmark, June 24 - 28, 2002). ITiCSE '02. ACM, New York, NY, 183-187.

[20] Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., and Tutty, J. 2006. Predictors of success in a first programming course. In *Proceedings of the 8th Austalian Conference on Computing Education - Volume 52* (Hobart, Australia, January 16 - 19, 2006). D. Tolhurst and S. Mann, Eds. ACM International Conference Proceeding Series, vol. 165. Australian Computer Society, Darlinghurst, Australia, 189-196.

[21] Wilson, B. C. and Shrock, S. 2001. Contributing to success in an introductory computer science course: a study of twelve factors. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education* (Charlotte, North Carolina, United States). SIGCSE '01. ACM, New York, NY, 184-188.

[22] Williams, G. C., Bialac, R., and Liu, Y. 2006. Using online self-assessment in introductory programming classes. *J. Comput. Small Coll.* 22, 2 (Dec. 2006), 115-122.

[23] Woit, D. and Mason, D. 2003. Effectiveness of online assessment. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA, February 19 - 23, 2003). SIGCSE '03. ACM, New York, NY, 137-141.

[24] Zhang, D., Zhao, J. L., Zhou, L., and Nunamaker, J. F. 2004. Can e-learning replace classroom learning?. *Commun. ACM* 47, 5 (May. 2004), 75-79.

Implementing a Contextualized IT Curriculum: Changes Through Challenges

Matti Tedre[1,2]
[1]University of Joensuu
Department of Computer Science and Statistics
Joensuu, Finland
firstname.lastname@acm.org

Nicholas Bangu[2]
[2]Tumaini University
Iringa University College
Iringa, Tanzania
firstname.lastname@tumaini.ac.tz

ABSTRACT

In this article we analyze the challenges that face IT education in a private university in rural Africa. Our analysis is based on a two-year ethnographic field study and action research project in a higher education institution. Our analysis reveals that student selection is contingent upon nationwide government decisions. We present that IT manufacturers' pricing policies are one reason for the digital divide between universities of the Global North and Global South. We consider aspects of collaboration between universities. We analyze the effect of organizational structure to the implementation of an IT program. We discuss students' preconceptions about IT as a field of study. We examine the difficulties concerning university-level staffing in developing countries. Finally, we summarize our analysis into seven normative lessons-learned, which we hope to be useful for other people undertaking similar projects in the developing world.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education, curriculum*

1. INTRODUCTION

In 2007 Tumaini University started a new kind of an information technology (IT) program—a contextualized one—that was tailored to meet the needs of Iringa region in Tanzania yet at the same time resonate with the curricula recommendations of the Association for Computing Machinery (ACM) and Institute of Electrical and Electronics Engineers' Computer Society (IEEE-CS) [14]. The first year of running Tumaini's B.Sc. Program in IT (BSC-IT) was spent resolving uncertain and unclear issues concerning the goals and objectives of contextualized IT education, and those steps have been documented earlier [16].

The second year of running the program saw a number of issues and threats that, with hindsight, are probably typical of a new academic program in a private university in a developing country. There are various reasons for the challenges we faced. Some issues derived from political decisions that affected the entire Tanzanian educational sector. Some were given rise by the global market economy, which often treats developing countries harshly, and which allows unfair treatment of developing countries by multinational corporations. Some challenges arose from differences in academic, communication, and bureaucratic cultures between partner universities. Some issues were home-grown, and can be attributed to the program being in an early, formative stage. Some tension derived from students' uncertainty about IT as a field and from difficulties in studying on university level. And some conflicts happened due to some staff members' lack of experience on university-level teaching as well as their unfamiliarity with standards of quality education.

In this article we outline some of the challenges we faced during our program development as well as some of our solutions. We detail some common hardships of IT education in a developing country context. Finally, we propose suggestions for coping with some issues in development of IT education in developing countries.

Koli Calling '09 October 29–November 1, 2009, Koli, Finland

2. RESEARCH METHOD

This paper reports an investigation and analysis of the educational, social, and cultural environment of a new educational program. As a combination of ethnographic research and participatory action research [4], this research focuses on exploring challenges and prospects that the Tanzanian context brings into the development of an IT program. Nuances of sociocultural interactions as well as many ethnographic observations are largely excluded due to the limited length of the paper, but a richer description and analysis of pedagogical issues in IT education in Tanzania can be found in authors' previous work [14]. Typical of ethnographic research [3], we aim at exploring local particulars, emphasize adaptability in the course of study, develop new concepts over the course of the study, and represent data mostly in natural language.

Reports of interpretive research are easily biased by personal opinions and positions [7]. Therefore, it is important to bracket our positions in the organization we study. This paper was written from the viewpoint of two people associated with the program: we write this paper qua Associate Professor (Head of the Program) and Provost (CEO) of the college—both doctoral degree holders. The Head of BSC-IT program was hired from outside the organization to run the program and to launch a continuous improvement pro-

cess within the program, and although he coordinates the BSC-IT program's development, he is currently working for a European university. The second author is the Provost of Tumaini: he initiated the BSC-IT program, chaired the design and implementation process of the program, and he continues to oversee the program's development. Through the experiences of these two key administrative people, we wish to bring forth a thick description of phenomena surrounding the program's formative years.

Our views about the program's development are necessarily biased by our positions and history with the program. In this paper we present the challenges and prospects as we see them, and our views surely differ from those of other stakeholders. To complement our lived experiences we use data collected from various sources: e-mails, student feedback, internal memos, seminar presentations, notes on discussions, and field notes. Those data sources are indicated in footnotes where appropriate. The data analysis was a basic qualitative data analysis where emerging themes, patterns, and signals were analyzed for resonance—or dissonance—with research literature [11].

3. IT PROGRAM AT TUMAINI

The B.Sc. program in IT is the newest Bachelor's level program at Tumaini University, Iringa University College. The program was started in September 2007. The IT program is based on six principles that have from the beginning steered the development of the program curriculum [14, 19]. The first principle, *context-sensitivity*, refers to the idea that each society, climate, environment, economy, and culture pose some unique challenges for IT professionals, and that local IT curricula should respond to those challenges [14]. The second principle, *problem-orientation*, refers to the typical constructivist approach of problem-based and project-based learning—that is, students work on authentic problems and reflect on the experiences they gain while working on those projects [9].

The third principle, *practicality*, refers to the idea that without practical training the graduates may not be able to work with the various hands-on tasks that are expected of them. The fourth principle, *interdisciplinarity*, refers to the combination of different computing curricula, as well as other subject areas, in the curriculum [16]. The fifth principle, *international recognition*, refers to the fact that in order for graduates to work in countries other than Tanzania and to continue their studies in international master's level programs, the program must resonate with the international standards of IEEE and ACM [1, 2]. The sixth principle, *basis on research*, refers to the need to base any revisions of the curriculum on rigorous research on-site.

Originally the IT program was an independent unit that responded directly to the university's top administration. After the program had been run only for half a year it was, however, accommodated within a newly formed ICT directorate (an independent unit of a smaller size than faculty) [16]. No more than half a year later, the IT program was re-allocated to newly formed Faculty of Science and Education.

During the academic year 2008–2009, BSC-IT program's teaching staff consisted of two tutorial assistants, assistant lecturer, ICT director, and one associate professor. The associate professor held a doctoral degree in computer science, the assistant lecturer finished his online M.Sc. studies in early 2009, and the other teaching staff members held B.Sc. or B.Tech. degrees in computing. In the end of the academic year 2008–2009 there were 27 second-year students and 30 first-year students—48 men and 9 women altogether.

4. CHALLENGES AND CHANGES

The challenges of the first year of operation were mostly about finding a fit between the ambitious but ambiguous plans for Tumaini's IT program and the actual implementation of the program [14, 16]. Practicality, problem-based orientation, context-sensitivity, interdisciplinarity, international recognition, and research-based program development are nice keywords but unclear and not easily implementable. Organizational matters were unclear during the first year of operation, and a lot of curriculum design was still needed [16]. In this section we outline seven most challenging issues we faced during the second year of operation.

4.1 Surprises in Student Selection

Student selection at Tumaini University is a complicated and long process, and although the rules and criteria are well established and rigid, the outcomes are, for external reasons, eventually unpredictable. In July 1st, 2008, after receiving the first list of entrance applications, Tumaini's Academic Board convened for student selection. That meeting took together the whole administration, deans, and department heads, and it lasted nearly eight hours. BSC-IT program was able to select 19 students from the first call. Those students had, generally speaking, impressive records in their high school A-level (advanced level) studies.

However, very soon after the selections to Tumaini were made and were submitted to Tanzania Commission for Universities (TCU) for processing, TCU's officials replied that they had assigned each and every one of those 19 students to a new national University in Dodoma[1]. UDOM was founded in 2007 by a governmental fiat, it hosted 7.000 students in 2008–2009 and is projected to have 16.000 students in 2009–2010 and 40.000 students in 2010–2011[2]. The current government tries to guarantee the success of their founded institution by channeling the best students and considerable academic resources there. The ambitious growth plans of the new national university pose a serious threat to other universities' academic programs in terms of staffing, student intake, and resource allocation. What is more, as we write this article, due to difficulties in TCU's joint application process and due to constant late announcement of the national loans board's decisions, the start of academic year 2009–2010 has been postponed by one month.

Government's decision on the student selection lead Tumaini University to have a second call for IT students, which attracted some 40 applications, out of which 25 met university's stringent admission requirements [18]. Because the quota of 30 students was still not fulfilled, the program administration announced an internal transfer call: Those students who had been accepted to Tumaini's other programs, but who met the admission requirements to the IT program, could transfer to IT. As students were aware that the national student loans board—which is the main sponsor of

[1]Announcement on Tumaini University web pages, retrieved November 11, 2008.

[2]Interview of Prof. Casmir Rubagumya, acting Vice Chancellor of the University of Dodoma. *The Citizen*, Special Education Issue, June 2, 2009:p.7.

most students—has a preference on science and technology fields, the remaining 5 student posts were quickly filled[3].

4.2 Trouble with Equipment

When the IT program was about to procure equipment for a new computer lab for the incoming group of students, the procurement staff ended up in a situation that is very familiar to Tanzanian ICT professionals. The price of equipment for the IT program's new computer lab was almost double the price that U.S.-based educational institutions would pay for the same equipment. And if hardware is expensive, original software is often even more expensive. One of the few exceptions to the rule is Microsoft, which, under its "unlimited potential" campaign sells operating system and office products for schools in developing countries for a few dollars per copy.

The Tanzanian government wants to encourage information society development by having a 0% import tax on computing equipment that comes for educational purposes. Regardless of that, computers are still more expensive in Tanzania than in Europe or the United States. The situation is the same throughout East Africa: In his closing speech of IST-Africa Conference 2009, the ICT minister of Uganda noted that East Africa will continue to increase computer equipment trade with Chinese manufacturers because in comparison to European and U.S.-based computer manufacturers, Chinese manufacturers can sell the same quality products for much lower prices. As mainstream computer manufacturers—such as Dell, Toshiba, HP, and Apple—do not offer the same level of support in developing countries as they do in Global North, it is hard to understand the differences between equipment prices.

The BSC-IT program already had a computer lab with Intel-PC / Microsoft + Linux computers, so it was decided that the second lab should contain Apple Macintosh computers. The idea was to teach the students to move flexibly between different brands and types of computer systems. That lab would also probably be most utilized, due to chronic virus problems on Windows-based computers in Africa. However, the prices stunned even Tumaini's seasoned IT technicians. Table 1 portrays some example prices of hardware and software from five vendors[4]: Apple web stores in the U.S. and Europe, and three major Apple resellers in Tanzania (Tan_1–Tan_3).

Table 1: Examples of Hardware and Software Prices on Selected Products

	U.S.	Europe	Tan_1	Tan_2	Tan_3
iMac 20"	$1199	$1556	$1650	-	$1700
iMac 24"	$1499	$1980	$2250	$2565	$2890
iWork '09	$49	$84	$245	$130	-
FinalCut 4.0	$199	$282	$595	$500	-
Aperture 2	$199	$282	$550	-	-
AppleCare	$169	$253	-	-	-

The first setup in Table 1 is a minimum setup (iMac 20" / 2GB / 320 GB) without extra software pre-installed. The second setup is suitable for video editing (iMac 24" 2.66GHz / 4GB / 640 GB, which was the highest performance model available in Tanzania). In the U.S. and Europe, one can purchase iWork and FinalCut Express as pre-installed options, but in Tanzania none of the vendors offered that option. When queried about the cost of AppleCare warranty, one of the Tanzanian resellers wrote, "*Unfortunately we are not permitted to sell Apple care in Tanzania and so we cannot extend the warranty beyond the standard one year.*"[5] AppleCare three-year warranty is available for additional $169 in the U.S. and for $253 in Europe. Simply put, in the poorest countries in the world, hardware vendors (Dell, Toshiba, Apple, and apparently all others) sell their products for higher price and poorer terms than anywhere else in the world.

What is more, when products bought from outside fail, "global" warranty contracts cease to apply in Tanzania: One IT staff member's Apple laptop broke, but even though the laptop was registered to the current owner in Apple's global database, Tanzanian Apple service did not offer warranty service because the owner did not have with him the original purchase receipts on paper. The happy Apple owner—with an 8-month old MacBook Pro laptop and a 3-year AppleCare warranty—had to pay the repair himself with no possibility for refund later. This interpretation of warranty terms was then confirmed, by the Nordic Apple service in Europe, to be correct[6]. The Apple service in Tanzania told that they require a written proof of purchase because "*How would we know it's not imported through gray markets*"[7]—whatever 'gray markets' mean in a global economy for visiting researchers who bring their laptops from their home countries. It seems a bit old-fashioned to ask today's cosmopolitan IT workers to carry paper receipts of computer purchase in their computer bags for three years.

When the Head of IT program contacted Apple headquarters, in April 2008, to query about the reasons for the high prices, his queries were directed to a high level Apple manager in Europe. Over a phone call the Apple representative explained that among other reasons, the higher prices are an outcome of small markets, remote location, and high risks (in, e.g., shipping and transportation—for instance, ships that sail past the Horn of Africa have high insurance costs due to frequent piracy incidents). Although the Apple representative sympathized with the plight of the BSC-IT program and took great effort to significantly bring the prices down, the prices still remained too high compared to other solutions. Although the Head of BSC-IT program recommended getting Apple computers, the ICT Director and top management ended up obtaining generic Intel-PCs running Windows XP and Office solutions, parallel with Ubuntu Linux[8]. Later, however, regardless of the high price, the BSC-IT program purchased one 24" top-end iMac computer with video editing software for multimedia editing purposes. That single Macintosh computer has been in constant use—often from five in the morning to midnight.

4.3 Vital Collaboration

International and national collaboration is even more important in developing countries than in industrialized coun-

[3]Transfer applications, September 23–October 6, 2008.
[4]1€= $1.4144. Examples are from March-June 2009. Apple web stores are store.apple.com and store.apple.com/fi
[5]Personal e-mail from Apple reseller in Dar es Salaam, April 16, 2008.
[6]Personal e-mail from Nordic AppleCare, March 16, 2009.
[7]Personal Communication, January 29, 2009.
[8]Personal e-mail from ICT Director, September 16, 2009.

tries. Tanzania's acute lack of funds and substance expertise can be countered by creative use of staff and student exchanges, by e-learning co-operation, and by joint project work with partner institutions. In the following subsection we describe three aspects of collaboration that we feel are crucial for a program's success: Extent and variety of collaboration, learning each other's standards, and developing commonly agreed channels and ways of communication.

4.3.1 Collaboration

Tight collaboration with other universities has shown to be the most important success factor for Tumaini's IT program. That is due to three main reasons: Firstly, student and staff exchanges offer Tumaini and the partner universities new chances for research and collaboration. Indeed, one of Tumaini's great strengths is in how we can offer visitors unique possibilities for research in the region. Secondly, Tumaini gets recognized experts to teach IT students short courses and Tumaini's staff members get international exposure and experience. Thirdly, Tumaini's students get the chance for one-semester exchanges, for short intensive courses in Iringa and abroad, and for continuing to M.Sc.-level studies abroad.

During the second year of running, the BSC-IT program joined three major projects. The first one, an EU-funded project coordinated by University of Joensuu in Finland, is a joint project between four European and five African universities, and it focuses on development of a joint ICT for Development (ICT4D) curriculum and learning material. The second one, also coordinated by University of Joensuu in Finland and funded by the Academy of Finland, is focused on improving the contextual impact of IT Education in Tanzania. The third one, funded by the OLPC (One Laptop Per Child) Foundation in the U.S. and coordinated by a group of Tumaini's IT students, aims at developing IT skills and independent learning skills of primary school children in Ukombozi village in Iringa—that project will take 100 XO-1 laptop computers to be used for educational purposes in poor Ukombozi Primary School.

In terms of distance learning, some of Tumaini's IT courses are fully online. For instance, a course on Cyber Law was given on-line, from Europe, by Mr. Andrew Mollel, who is an expert on Tanzania's electronic legislature. But Tumaini's slow network connection made it slightly cumbersome for the law lecturer to organize his own timetable for the course. In that course Mr. Mollel had to access Tumaini's Moodle in the night between 22:00 and 08:00 when Tumaini's computer laboratories are empty and the college's satellite connection has a light load. Mr. Mollel reported that during daytime the satellite link could not sustain a connection stable enough and fast enough for productively working with Moodle.

But international projects are not that easy to set up in a small unknown university in rural Tanzania. Some applications to major funding agencies were returned without review, and some agencies stated, among other notions, that in Tanzania they work exclusively with the University of Dar es Salaam[9]. This focus of foundations is understandable, as University of Dar es Salaam (UDSM) is the largest university in Tanzania. However, the size and complexity of UDSM has also made it notorious for frequent involvement in corruption scandals.

[9]Personal e-mails, July 2008.

4.3.2 Bureaucracy

Although inter-university collaboration has been essential for success, it has also brought upon us new kinds of burdens. In January 2009, the college hosted the kick-off meeting of a large EU-funded educational project, where the BSC-IT program plays a significant role. However, neither the IT program administration nor the Northern partners were fully familiar with the EU procedures and standards of accountability, which caused significant trouble to both partners of the collaboration. After the Northern partner promised Tumaini that the funds for organizing the meeting will be transfered to college in one week's time, the administration agreed to use college's (very limited) funds for organizing the workshop, paying transportation, paying hotel fees, and so forth. However, very soon both sides of collaboration ended up mired in overly rigid EU bureaucracy, which delayed the payments to the extent that the IT program administration and professors in Tanzania had to use significant amounts of their own money to pay off some of the debtors while waiting for the EU funds to arrive. That also severed some personal relationships with local collaborators.

The main cause for friction was that all kinds of EU regulations were discovered one after another, so that new documents were requested from Tumaini every other day. For instance, Tumaini's employees are paid a monthly salary and they are not required to keep track of their working hours; the college's employees are given pre-paid vouchers for their work phones but they are not requested to keep track of their phone calls; the college does not have fixed rates for rental of facilities; and it is not a part of the college's standard practices to record the room numbers when one is accommodated in a hotel. All these, and many other checks and proofs were requested, in accordance with EU regulations, after the workshop. Eventually the administration had to tell the Northern partner that the administration refuses to fabricate phone call lists, hotel room occupancy lists, and other required evidence[10]. The aftermath of the workshop took Tumaini's administration numerous work days and unnecessarily strained the relationship between the two partners.

Developing countries are often criticized for overly bureaucratic procedures [10], but our experience is that EU has a similar level of bureaucracy—and unlike the African features of bureaucracy, the EU ones cannot be bypassed. For instance, the application process for EU visa for one of Tumaini's lecturers in December 2008 was a Kafkaesque experience for everyone involved. Whereas the African partners should indeed improve some standards of accountability, the European and U.S.-based partners should understand that accountability is not a one-way street: Ian Smillie wrote aptly that 'accountability' is a word that "*rolls easily off Northern tongues, as though it were a one-way concept. It could, and perhaps should be asked who holds* [the Northern partner] *accountable for inexplicable delays, for rigid reporting requirements and sudden changes of policy*" [12, 62]. It is very important that all the policies, procedures, regulations, and standards are made clear from the beginning of collaboration.

4.3.3 Communication

Tumaini University has, for several years, aspired to im-

[10]Personal letter to the partner, January 27, 2009.

prove the flow of information within, from, and to the university. An evaluation of the University as a high performing organization, conducted in 2007 by Maastricht School of Management, found that communication glitches constitute one of the major weaknesses of the university. The glitches have been a source of frustration to staff, visiting researchers and to collaborative links of the university. It was considered imperative that this weakness be effectively addressed in the shortest possible time, and a number of improvements are being implemented.

Firstly, we want to channel information flows from informal "coffee-room" channels and unorganized bulletin board announcements into an e-bulletin board system. We hope that this change will ensure that public announcements will reach the whole university effectively and in a centralized manner. Secondly, by establishing a position of an international coordinator, we want to centralize communication with our incoming visitors, visiting researchers, and exchange students. We hope that this change will reduce the amount of overlapping work between coordinating staff, reduce gaps between responsibilities, ensure quick and reliable communication, and enable quick flow of information from the university to our visitors. Third, we want to improve our internal communication from a mixture of email, letters, and memos to e-mail only. Meeting that goal, however, requires a very reliable e-mail system which is able to keep a "paper" track; and it also requires a shift in attitudes of users in terms of trust towards a paperless system.

4.4 Organizational Adjustments

As we noted before, within about a year the BSC-IT program at Tumaini saw three very different positions in the university's organizational structure. Originally the program enjoyed a high degree of independence and shallow bureaucracy, which were slightly changed at the forming of the ICT Directorate. In September 2008 the BSC-IT program was located, with the Department of Mathematics Education, under a newly formed Faculty of Science and Education. See Figure 1 for an illustration of the three organizational situations where the BSC-IT program was located during 2007–2009.

Throughout the organizational changes, the BSC-IT program sustained significant freedom in terms of program development, program implementation, and course design and implementation. In the course of time, the increasingly tight conformance to university's organizational structure affected flexibility, decision-making, and accountability. In the beginning the program enjoyed considerable flexibility in terms of fast decision-making, quick and working communication, and light organizational structure—the further developments, however, significantly increased red tape, levels of bureaucracy, and rigid procedures to follow. On the other hand, those developments also enabled better evaluation and control mechanisms on the program. Those changes also lessened the burden the BSC-IT program caused to the top administration.

When a new Faculty of Science and Education was formed, the new faculty structure introduced a new Dean of Faculty, who holds a Master's degree in education, and who is very well connected with all stakeholders in the educational sector in Iringa region and in Tanzania. The effects of this organizational change on BSC-IT staff's workload were mixed. On one hand, changes in hierarchy brought in another level of bureaucracy—the faculty level—which made procedures more rigid and which complicated decision-making. On the other hand, many departmental duties as well as human resource management tasks were moved to faculty level, which eased the administrative burden. In addition, the Dean of Faculty delegated to the Head of IT program considerable freedom and responsibility on IT program administration on matters of IT teaching, which enabled quite some flexibility in decision making. All in all the organizational shifts did not change much the amount of work done in the program. See Table 2 for a summary of changes that organizational changes brought about.

However beneficial or disadvantageous the organizational changes will be in the long run, the frequent changes in organization made it difficult for the BSC-IT program staff to know their place in the university organization, to understand and follow the management and reporting chain, and to handle information flows properly. More often than not it was unclear to whom different requests, forms, reports, or memos should be addressed. With hindsight, many of the organizational difficulties and communication glitches could have been avoided if more departmental meetings had been held. But difficulties and clashes between people in the first departmental meetings had led to abandoning that practice. Later, though, on administration's request monthly departmental meetings were started again.

4.5 Conflicting Conceptions of IT Studies

The students' determination concerning the quality of their education, combined with their simultaneous, significant lack of technological literacy, has caused quite some friction in the program [15]. Students have their own ideas about what they need, and they can sometimes quite vehemently advocate those ideas. It has taken quite some effort to explain to students some basic facts about computing education in general, and IT education in particular. Below we list a number of ideas, questions, or complaints that students have posed, and our responses to those.

1) "*Our program is not theoretical enough.*" Or, in one student's words, the program is "like an advanced secondary school". A number of students who have talked with computer science students at University of Dar-es-Salaam and Ruaha University College have admired the theoretical orientation of those programs, and wished for more theoretically oriented perspective to the BSC-IT program too. We have found that it is helpful to thoroughly explain the differences and characteristics of computing fields—electrical engineering, computer engineering, computer science, software engineering, information technology, and information systems [2]. It is fruitless to compare the field of theoretical computer science with the field of information technology [13, 17].

2) "*Our program does not prepare us to work in IT fields.*" In that statement students use the phrase "IT fields" to implicitly refer to some specialized computing field, such as fingerprint recognition, color research, or artificial intelligence. We have countered this comment by illustrating students the width of the field, the vast number of fundamental technologies of computing, and the variety of branches in computing [5, 6]. In the field of IT a degree does not make one "ready". A degree in IT prepares one for learning their job quickly and for keeping up with the constant changes in the field.

3) "*Our course on topic x differs from the same course in*

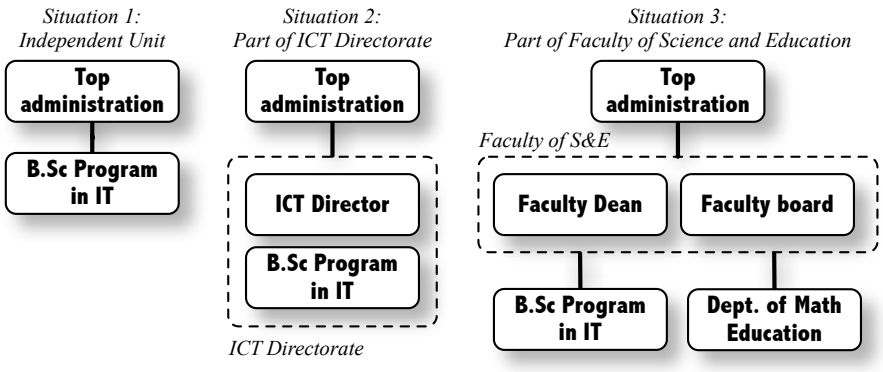


Figure 1: Three Succeeding Organizational Situations of BSC-IT Program

institution y." These objections sometimes arise when Tumaini's students compare what they have studied with what students in other institutions have studied. For some reason, there is a feeling that all students in computing fields should have the same set of courses with the same content. We respond to this by explaining that there are neither worldwide adopted IT / computing syllabi, nor are there standard contents for specific courses. The ACM/IEEE curriculum guidelines, such as CC2001 and IT2005, offer good directions, but intentionally leave a lot of room for local considerations ([1, 12–13], [2], [14]).

4) "*An IT specialist should know x, y, and z, but those topics are not taught in our program.*" This argument often arises when students have been reading job advertisement and see the highly specialized skills that job applicants should possess. Substitute x, y, and z with any implementation-specific terms such as "Oracle", "C++", and "Cisco networks", and someone in our IT program has probably raised a question about their lack in the syllabus. Firstly, in our IT program we explain that a B.Sc. degree is not supposed to produce specialists but generalists. Master's-level programs and doctoral programs are meant for specializing students. Second, we stress that life-long learning and self-learning are quintessential in the field of computing. New technologies and techniques require constant attention to learning new things. We try to introduce and instill in our students the idea that learning about IT *begins* at the university and continues as the students leave the place.

4.6 Difficulties with Staffing

Like other rural universities in Africa, Tumaini has a persistent problem with attracting formally qualified staff. However, having formal qualifications is only one criterion of competence—other criteria include things like work experience, deep knowledge about IT topics, commitment, and pedagogical deftness. In the beginning, Tumaini's IT program was able to hire three B.Sc./B.Tech. degree holders who had 3-6 years of work experience from the field of IT [18]. In addition, the university got one Ph.D. holder to head the new program. The second year's staff hiring, however, did not go as smoothly as the first round did.

4.6.1 Recruitment

The original plan for academic year 2008–2009 was to hire three more staff members, preferably M.Sc. degree holders[11]. There were a number of qualified applicants, and preliminarily the idea was to hire one dual degree (M.Sc. and M.Eng.) holder from India (a wife of a current Tumaini employee) and a Tanzanian M.Eng. degree holder who had received his degree from Europe. However, visa problems caused the Indian applicant to cancel her coming, and after that financial concerns at Tumaini led to the college freezing all new staff hiring[12]. That action caused a conflict between the program administration and the Tanzanian applicant, who had already turned down some other offers in favor of Tumaini's offer. Furthermore, one IT teacher's return from study leave was mired in an argument over work contract terms. And finally, one of the key people in the program, the ICT Director, left for a two-year study leave in a European institution. Suddenly the situation was that instead of three more teachers, the IT program had double the number of courses to teach, but one less teacher. Even further, one of the original teachers moved from IT education to ICT support department[13]. The situation was slightly helped by a long-time IT staff member of Tumaini who was still finalizing his continuing studies, but who took part-time teaching in the program.

The quite dire situation with staff eased temporarily in Spring 2009. Two of Tumaini's IT staff members, who had been with Tumaini University almost from the founding of the university, returned from their study leaves—now holding B.Sc.(Hons.) and M.Sc. degrees in IT. They were able to undertake a good portion of the program's teaching, and they were able to teach courses according to their own specializations. In mid-semester, though, the situation changed drastically when one of the IT teachers ended in a clash with IT students and with program management, and after a series of fierce conflicts abruptly resigned from teaching any of the four IT courses he was giving[14]. After almost a month of resolving the situation and reorganizing teaching, two part-time teachers were called in, and one course was undertaken by program management—which froze down many other vital activities, such as outreach project work and research activities. The crisis took nearly two months to fully resolve,

[11] Minutes of BSC-IT Planning Meeting, April 8, 2008.
[12] Personal e-mails, September 2008.
[13] Personal e-mail, July 30, 2008.
[14] Letter to management, April 18, 2009.

Table 2: Effects of Organizational Situation to BSC-IT Program

	Independent Unit	**ICT Directorate**	**Part of a Faculty**
Bureaucracy	Light; Only inter-department functions require paperwork	Intermediate; Frequent communication required between BSC-IT Program and ICT Directorate	Burdensome; Delays due to paperwork, rigid procedures, and difficult access to key people
Decision-making	Flexible / Three levels: BSC-IT program, top management, and academic board meeting	Flexible / Four, partly overlapping levels: BSC-IT program, ICT Director, top management, and academic board meeting	Rigid / Four levels: BSC-IT program, Faculty meeting or Dean, top management, and academic board meeting
Clarity	Some ambiguity about responsibility and procedures	Some ambiguity about responsibility and procedures	Clear location in the organization; clear responsibilities and procedures
Accountability	Low; Decision-making happens within the unit with few checks	Low; Decisions are made among ICT staff	High; Decisions have to be justified and defended on several levels
Vulnerabilities	High dependency on key people, lack of accountability, human relationships, lack of insight into the big picture	Dependency on key people, resource competition, narrow view of development	Resource competition, conflicting interests, conflicting views on development, communication breakdowns
Support	Very strong, but dependent on management's will and commitment	Strong, but vulnerable to conflicting interests of ICT support and IT department	Strong, but dependent on Faculty Dean's commitment and qualities
Administration	Burdensome, all administration, technical development, and human resource management are done within the program	Burdensome, administration is distributed between two small units	Intermediate, some administrative duties are delegated to faculty level. Some centralized support functions.

and seriously impeded all activities in the IT program.

4.6.2 *Varying Perspectives to Teaching*

There is a variety of perspectives among staff members concerning quality of teaching. Many teachers at Tumaini, including the IT program, share the constructionist view of the teacher as a "guide on the side", as well as the central ideas of practical and problem-based learning. However, in a departmental meeting three teachers noted that the principles concerning teaching at Tumaini's IT program are very different from those in the University of Dar-es-Salaam (UDSM)[15]. Those teachers specifically mentioned three differences between UDSM's Computer Science program and Tumaini's IT program.

Firstly, they argued that at UDSM professors and lecturers can sometimes be absent for extended periods of time without repercussions. Secondly, they argued that at UDSM students have to take responsibility of their own studies instead of the "spoon-feeding" practices at Tumaini's IT program. Third, they argued that at UDSM students are not encouraged to visit—or are even prohibited from visiting—their lecturers' offices or asking questions about course contents. One teacher proposed that a suitable mode of education is one where the lecturer comes to the lecture hall, presents the lecture, and leaves. After some discussion he conceded that "maybe we can allow five questions at the end of the class"[16]. At the end of the second year another teacher insisted that giving only five lectures out of thirty projected lectures constituted quality teaching, and refused to consider arrangements for delivering the remaining course contents to students in the following semester[17]. That crisis too led to a severe clash between IT students, the teacher, and administration; at one point students issued a strike notice unless the problem with the course is resolved.

Discussions with teachers revealed that some of them regarded the UDSM modus operandi to be more desirable than the strict rules currently at place at Tumaini. In addition, some have found it undesirable that they have to show up at the college also outside lecture hours; one of the teachers asked sarcastically, "so are we expected to stay on campus even when we don't have anything to do?"[18], which led to a rather unpopular reassessment and reassignment of departmental duties[19]. In a workload evaluation it was found out that departmental duties were distributed very unevenly, yet some staff members were reluctant to assume any responsibilities that were not directly teaching-related. Some teachers did not consider technical and administrative tasks, such as IT program's website management or coordination of internships, to be a part of their work contract. Again, the issue ensued in a conflict that went all the way to the college's personnel administration officer, who ended up distributing clear instructions for work-related duties.

Finally, we learned that feedback from students to teachers must be thought very carefully before implementing it. At the end of the first year students returned their annual feedback, where they commented on what is good in the pro-

[15]Departmental meeting, June 24, 2008, 14:00.

[16]Direct quote: Ethnographic field notes / personal communication, September 18, 2008.

[17]Ethnographic field notes / personal communication, June 17, 2009.

[18]Ethnographic field notes / personal communication, September 18, 2008.

[19]Internal memo, September 19, 2008.

gram and what could be developed. In addition, students evaluated the courses, and those evaluations were submitted anonymously and privately to the teachers of each course. In many occasions the program administration emphasized that the student feedback is for teachers' own personal development only, and that this kind of feedback is not meant to be any kind of a professional evaluation. However, the feedback caused one of the teachers lashing out, calling this kind of feedback mere "rumors" and calling the students lazy[20]. The idea of basing professional development on student feedback has to be grounded and marketed in a culturally suitable way.

Students' attitudes towards quality of education have been critical, yet mixed. We see a clear trend that those teachers who utilize problem-based learning, who are not afraid to work on the level of students, and whose pedagogical views are at least implicitly constructivist are highly respected and liked among students. However, despite the alleged "high power-distance" culture in Tanzania [8], students have openly opposed teachers who elevate themselves or disrespect students, who stick to a simple lectures–and–rote memorization pedagogy, or who otherwise cannot attain a satisfying level of quality in education. We presume that during their own studies each of the teachers have acquired an idea about how quality education should be done, and they let those ideas guide them in their own teaching. Modernizing outmoded pedagogical views is a difficult but necessary task in our college.

5. LESSONS LEARNED

We have collected here lessons that we learned during two years of program implementation, as well as our suggestions for coping with some difficult situations when developing a new IT program in a developing country. Although the lessons seem intuitively appealing, one ought to keep in mind that the recommendations we make are probably not generalizable to all developing countries.

Lesson 1: Prepare for Flexibility in Student Selections

In Tanzania, creating a good public image for IT program is vital for attracting applicants—but that might not always be enough. Political motivations and changes in government's regional and educational policies may have significant influence on student intake. It happens every year at Tumaini that at the beginning of the semester only a fraction of incoming student quota is filled, and additional calls have to be made. Around two or three weeks after the beginning of the semester, the final list of new students is formed and first-year classes may begin.

Lesson 2: Do Budgeting Locally

When planning an IT budget in a developing country, budgeting must be done locally, in local terms, and using local knowledge on prices, procedures, constraints, and quirks of procurement. Prices for equipment in developing countries are higher than prices for the same equipment in the Global North. From our perspective, developing countries need much less stripped-down "poor-country versions" of equipment than they need fair pricing, fair terms of trade, and fair terms of warranty. However, as long as the current situation prevails, budgets must be based on good local knowledge about prices, as well as on knowledge about what can be procured or manufactured in the region, what must be bought from Dar es Salaam, and what has to be imported.

[20]Group e-mail, June 23, 2008.

Lesson 3: Make Rules of Collaboration Clear

The procedures for managing a project throughout its lifespan—planning, implementing, monitoring, evaluation, and reporting—differ between many developing countries and industrialized countries. Procedures for monitoring, auditing, and reporting are often different, and sometimes there may not be qualified people to do those activities. Although those procedures often vary between grant agencies and funders, the differences are pronounced in international collaboration projects. In order to save time, money, and frustration, it is important to make the rules of collaboration crystal-clear from the very beginning. Unless it is absolutely necessary, the developing country partner should not be required to function as a provisory funder. If that is necessary, the timetables for remuneration as well as the procedures and format for expense claims must be made clear before any money is used.

Lesson 4: Start Flexibly

In the case of Tumaini's BSC-IT program, organizational rigidity and bureaucracy grew gradually over time. The free and flexible environment during the first year of the BSC-IT program contributed greatly to early shaping of the BSC-IT program. That light organization, however, lacked clear accountability, bypassed established procedures, and caused friction and stress with the rest of the university organization. Although it is a good idea to start flexibly, too much flexibility in one function or department of the university causes burden in other functions or departments.

Lesson 5: Educate Students About the Goals of Their Studies

IT is a broad field, and already when students apply to the program, they should understand the focus of the program. If students associate the term 'IT' with software engineering or with theoretical computer science, an information technology program will not meet their expectations. However, when students apply to university, they rarely have enough knowledge about computing to do an informed choice between computing fields. Therefore, at Tumaini, we begin very early to educate students about computing fields and their foci[14]. We are also underway making an information package for our website about the different computing fields.

Lesson 6: Recruit Staff Early, Don't Underestimate

Staff recruitment is hard, and a rural university might not be the first choice of job seekers. Therefore, it is important to be very active in recruitment and recruit internationally. But uncertainty about the college's finances means that promises should not be made before the final budgets are set, which easily deters the best applicants from choosing Tumaini among their job offers. When recruitment needs are being evaluated, it is important that the projected number of staff members is not set too low. If the number of hired staff members is kept to the minimum, then quality of the program's education, research, and outreach are jeopardized if even one staff member leaves the college.

Lesson 7: Make Short-Term Practical Plans

Financial, political, and socio-economic contingencies make long-term planning nigh impossible at Tumaini. Staff turnover is high due to continuous education of staff as well as to natural turnover of employees. Recruitment of new staff is complicated. University's finances are scarce and somewhat volatile, an the finances depend on the national loans board's payment schedule to students. The project-based nature of foreign grants and funding also hampers long-term planning. It seems that in order to accommodate the difficulties in practical long-term planning, East African organizations eagerly introduce policies, visions, and strategies—which may or may not be used to guide actions in the future.

6. ACKNOWLEDGMENTS

This research was partly funded by the Academy of Finland grant 128577, "*Improving the Contextual Impact of ICT Education in Southern Tanzania: Engaging Stakeholders Towards Innovation*," led by Prof. Erkki Sutinen, University of Joensuu, Finland (2009–2012). We wish to thank Mr. Lotti Chuma, who is the procurement officer at Tumaini University, for his relentless search for price data. We also wish to thank Mr. Deodatus Mogella for his help with data collection.

7. REFERENCES

[1] ACM Computer Science Curriculum Committee. Computing curricula 2001: Computer science, 2001.

[2] ACM Information Technology Curriculum Committee. Computing curricula: Information technology volume, 2005.

[3] M. Agar. Ethnography. In N. J. Smelser and P. B. Baltes, editors, *International Encyclopedia of the Social & Behavioral Sciences*, volume 7, pages 4857–4862. Elsevier, Oxford, UK, 2001.

[4] P. Atkinson and M. Hammersley. Ethnography and participant observation. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 248–261. SAGE, London, UK, 2nd edition, 1994.

[5] P. J. Denning. Great principles of computing. *Communications of the ACM*, 46(11):15–20, 2003.

[6] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[7] C. Ellis and A. P. Bochner. Introduction: Talking over ethnography. In C. Ellis and A. P. Bochner, editors, *Composing Ethnography: Alternative Forms of Qualitative Writing*, pages 13–48. AltaMira Press, Walnut Creek, CA, USA, 1996.

[8] G. Hofstede. *Cultures and Organizations: Software of the Mind*. McGraw-Hill, New York, NY, USA, 1997.

[9] D. H. Jonassen. Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4):63–85, 2000.

[10] E. Paloheimo. *Tämä on Afrikka*. WSOY, Helsinki, Finland, 2007.

[11] G. W. Ryan and H. R. Bernard. Data management and analysis methods. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 769–802. SAGE, Thousand Oaks, CA, USA, 2nd edition, 2000.

[12] I. Smillie. *Mastering the Machine Revisited: Poverty, Aid and Technology*. Practical Action Publishing, Warwickshire, UK, 2000.

[13] M. Tedre. Computing as engineering. *Journal of Universal Computer Science*, 15(8):1642–1658, 2009.

[14] M. Tedre, N. Bangu, and S. I. Nyagava. Contextualized IT education in Tanzania: Beyond standard IT curricula. *Journal of Information Technology Education*, 8(1):101–124, 2009.

[15] M. Tedre and M. Kamppuri. Students' perspectives on challenges of IT education in rural Tanzania. In P. Cunningham and M. Cunningham, editors, *Proceedings of IST-Africa 2009 Conference*, Kampala, Uganda, May 6th–8th 2009.

[16] M. Tedre, F. D. Ngumbuke, N. Bangu, and E. Sutinen. Implementing a contextualized IT curriculum: Ambitions and ambiguities. In A. Pears and L. Malmi, editors, *Proceedings of the 8th Koli Calling International Conference on Computing Education Research*, pages 51–61, Lieksa, Finland, November 13th-16th 2008 2009.

[17] M. Tedre and E. Sutinen. Three traditions of computing: What educators should know. *Computer Science Education*, 18(3):153–170, 2008.

[18] Tumaini University, Iringa University College. Academic prospectus. Prospectus Series, 2007–2008 and 2008–2009, Iringa, Tanzania, 2007.

[19] M. Vesisenaho. *Developing University-Level Introductory ICT Education in Tanzania: A Contextualized Approach*. PhD thesis, University of Joensuu, Department of Computer Science and Statistics, Joensuu, Finland, 2007.

Communicating with Customers in Student Projects: Experimenting with Grounded Theory

Ville Isomöttönen *
Department of Mathematical Information Technology, University of Jyväskylä, Finland
ville.isomottonen@jyu.fi

Tommi Kärkkäinen
Department of Mathematical Information Technology, University of Jyväskylä, Finland
tommi.karkkainen@jyu.fi

ABSTRACT

The study provides a grounded theory based conceptualization on students' communication with customers in the context of a software engineering capstone course. The course in question is the one-semester capstone project course (*Software project* `TIES405`) at the Department of Mathematical Information Technology, University of Jyväskylä (JYU/MIT). The results indicate that the students — novices at software projects — demonstrate a communication barrier towards customers as they enter a realistic software development context. Underlying causes of this main theme are presented and discussed, while the work also dissects the use of grounded theory.

Categories and Subject Descriptors

K.3.2. [**Computers and education**]: Computers and Information Science Education—*Computer Science Education*

General Terms

Human Factors, Theory

Keywords

Software Engineering Education, Capstone project, Communication skills, Grounded Theory

1. INTRODUCTION

Today's engineering education pays more and more attention to communication skills [19]. Good communication skills contribute to career development [22] and are expected from a professional [14]. The importance of communication skills has also been recognized in software engineering education (SEE) community. Freeman et al. [6] included the subject in essential elements of SEE in 1976 and the subject is found in SE2004 recommendation [23], under the knowledge area of professional practice, titled as *communications skills (specific to SE)*. Recently, courses in software engineering programs are being devoted to people issues that address communication skills, e.g. [25].

While the need to enhance students' communication skills is recognized in SEE community, questions arise: What are the dimensions in students' communication difficulties that should be known and addressed by teachers? What are these dimensions with regard to various course contexts and levels of computing education? To answer these questions the research needs to move from general observations to developing of analytic and explanatory knowledge — to uncovering the essential underlying mechanisms of a studied social object. The article aims to characterize students' communication skills in a precise and an analytic way — by systematic conceptualization. The research motivation is to achieve explicit knowledge of the students' communication difficulties. This knowledge is assumed to benefit the teaching in the course, cf. the use of phenomenography in computing education research [2].

A considerable communication challenge for the `TIES405` students is communication with real customers, which is the focus of this article. The work started based on the observation that the students demonstrate a communication barrier towards customers as they undertake the project work, and the need to support students' communication with customers is a constant concern in the teaching. While the teachers had lots of word-of-mouth knowledge on the students' communication challenges, the aim here was to systematize this knowledge by tracking the most important teacher observations and by analyzing the students' written course experiences.

The conceptualization is based on Grounded Theory (GT), a theory generation process originated by Glaser and Strauss in Discovery of Grounded Theory [10]. GT has found its way to SE research, e.g. [11] [4], as well to computing education research, e.g. [17] [12]. As noted in [1], it is important to tell which approach of GT is followed. This work is based on Discovery of Grounded Theory and Glaser's ensuing elaboration on the method [8] [9]. The aim of this study is to generate theory instead of using GT procedures merely for data analysis.

2. RESEARCH METHODOLOGY

2.1 Research context

The project course at JYU/MIT has a twenty-year history[1]. The course is taken after finishing most of the major subject courses in the bachelor studies. The students have

*Corresponding author

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

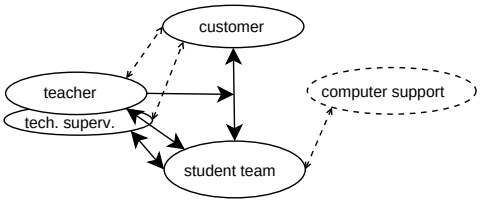


Figure 1: Interactions between project stakeholders

typically taken programming courses, algorithms and data structures, basics of databases, basics of web technologies, object-oriented analysis and design, an introductory course on software engineering, and some courses on math. Majority of the students do not have or have little previous software project experience. Staff members select the project groups of four students in each. They avoid assigning friends to the groups to create a communication challenge, and to avoid the emergence of the groups of two pairs or 1+3 compositions. The aim is also to have diversity in the project groups based on the students' study history and hobbies.

Interactions between project stakeholders are illustrated in Figure 1. A teacher (a staff member) is in charge of project supervision, focusing on process issues, whereas a senior student is responsible for technical supervision. The senior student is paid for the job (hourly wages) and expected to know the programming languages and development tools used in the project he/she is assigned to. The department's computer support is available for the student group who is responsible for contacting the support when necessary. All the stakeholders except the computer support meet in the project meetings which are arranged at least every other week. Meetings are not the only communication medium but informal communication takes place should the need arise (face to face, email, phone). As noted, in this work the research interest is in the interaction between the student group and the customer. The students do not take (consciously) part in the research that is reported here.

The students are expected to take lots of responsibility of their own work. In order to support the students' autonomy, each group is provided with a workroom. The course work with real project issues and real expectations on the project outcome mean a high work load for the students whether the customer is an external organization (which is preferred) or a university unit. With the one-semester period and real customers, the main learning objective is not the learning of particular by-the-book software process or engineering practices per se. Instead, the project groups start producing software with the competence they have and learn how to manage and finish the project.

2.2 Research method

At JYU/MIT, the project course research has been an ongoing process closely related to the first author's course development work during 2005-2008. The course development work aimed to notice the key areas enabling the student project course with real customers in one semester. Several issues entered the focus of this exploratory research, the students' communication with customers being one of those.

[1]Over 160 projects since 1995: `http://www.mit.jyu.fi/palvelut/sovellusprojektit/toteutetut.html`.

As noted, this study is based on Glaser's and Strauss' Discovery of Grounded Theory (DGT) [10] and Glaser's elaboration on the method [8] [9]. DGT defined a theory as a process. It suggested that a theory is generated from data and is then assumed to suit its supposed uses. DGT conceptualized a theory generation approach that differed from those using deduction and verification. It is a rigorous method with a systematic coding and analysis, but the rigor is here not what quantitative methods accomplish. The emphasis remains on thinking and creativity of an analyst. As noted in DGT, following GT does not guarantee a same result from two analysts [10, p. 103].

GT's on-going nature was considered suitable for the project course context where comparison groups follow one another. Instead of a fixed conceptualization, the research here provides snapshots of the process and reports on theorizing the researchers are confident with. The GT process was based on the constant comparative method outlined in DGT [10, pp. 101–115]. The use of the method is presented in Section 4, being better illustrated after providing the results.

The heart of an emergent grounded theory is the core category, a concept which provides context for other interrelating categories. An example of how the work followed Glaser's GT approach concerns the role of the core category. According to Glaser, the core category is a finding that is "relevant and problematic for those involved" [8, p. 93]. The core category has to emerge, and the analyst must become sufficiently confident with it, and then elaborate it to generate an analytic theory, see for example [9, p. 75].

2.2.1 *Data sources and schedule*

The data sources consist of teachers' observations and documented data. The former consists of word-of-mouth knowledge on communication issues that could be tracked to specific discussions among the teachers, and the teachers' direct observations that could be tracked to specific cases (projects).

The documented data encompasses three sources: 1) The course development plan which is based on the authors' and project teachers' experiences. The document theorizes the course context at a practical level focusing on guidelines for the teachers and the students, including notes on communication issues. The document was written during the first half of the course development work mentioned above. 2) The course evaluation statements the teachers write for projects (this is detailed when introducing the core category in Section 3). 3) The students' personal course experiences found in the project reports that the students write at the end of the projects.

The student project reports were sampled from years 2000-2007. The reports contain experiences of 121 students. The length of a student's written course experience in a project report ranges approximately from a third of a page (*A4*) to two and half pages of text. The projects were selected with the aim of providing a rich view on the projects. They involved both in-house and external customers as well as participation of seven different teachers.

The project reports are public documents. They are inspected by the project stakeholders and available for subsequent project course attendees as a reference material. In all cases the documents are checked by the customer to not include any confidential information. The projects with external customers involve an agreement to use project out-

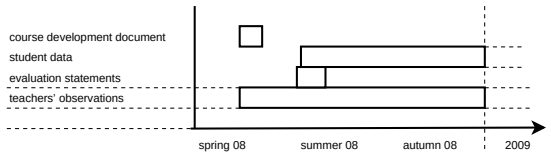


Figure 2: Research schedule

comes in research. Also, at the time of writing this article, the university research policy accepted the use of the project reports for a research purpose. The data was made anonymous for any reporting during the work.

The course context is often quite different for each project group. For example, the size of a project organization varies due to the varying number of customer representatives and end users involved, software domains and problem domains vary because the customers represent a variety of areas in life. Such temporary nature of a single project[2] is further emphasized in that the students undertaking the course do not share previous projects. Often, they do not know the people of the project beforehand. The work is thus not merely about applying grounded theory on a single case but the data represents many meaningful comparison groups (variance) in a single substantive area (a variety of capstone project instances in software engineering education).

Figure 2 depicts the schedule of the research process. The research interest started from the teachers' observations which are far reaching. The course has a twenty-year history and the teachers' word-of-mouth knowledge thus originates from a very long period. The actual analysis took place by returning to the documented course development data from the viewpoint of this study and by starting to track the most relevant observations in the teachers' word-of-mouth knowledge (observations) at the end of the spring 2008. This was complemented by analyzing the course evaluation statements and the documented student data during three months in the summer 2008. The findings were further explicated and re-analyzed in the autumn 2008. The beginning of the year 2009 was about writing and minor re-analysis. The long period is explained in two ways: firstly, the work was done irregularly besides teaching, and secondly, generating GT turned out to be a slow and difficult process. This was not due to amount of coding but the efforts to make sense of the data (analyzing and theorizing).

2.2.2 *Analysis techniques*

In this section, the analysis is described from a technical point of view whereas Section 4 is meant for illustrating it from a methodical point of view. The coding with the course development document was done by underlining the areas of interest and writing concepts down to the document's margins. Most important points in the teachers' word-of-mouth knowledge were tracked to specific discussions among the supervisors, or project instances which they related to. It was first felt that it is sufficient to keep this tracking information in memory. The analysis on documented student data was conducted by reading through the data while picking up all the relevant "something is being said here" points. Because this was the most extensive data source, the coding was managed by writing the key points down to a separate booklet with references to the original data. The concepts were identified and highlighted in the booklet notes.

Integration of the concepts emerged from the teachers' word-of-mouth knowledge. It was also allowed to continue as soon as the documented data was first skimmed through. The notes written down during the coding of the student data included integrative statements that took account of the other data as well. Hence, the observations (word-of-mouth knowledge that was tracked) and the concepts from the course development work were transferred to these notes. When starting to analyze the student data, drafting of the emergent theory, a consistent overall conceptualization, started also. All the data sources provided input here. A whiteboard and text documents were used for the drafting which in a technical sense was about preparing textual formulations and drawing diagrams.

In the re-analysis phase (autumn 2008), the emerging theory was textually formulated several times. At this point, the teachers' observations were yet extracted from the notes and textual drafts to a separate text file with the tracking information. It was felt necessary to explicitly maintain the origins of each observation; As the work was not a continuous process due to the analyst's (first author) other activities, some of the tracking information, that was first kept only in memory, had to be reworked here.

Overall, the above techniques highly overlapped. The writing and drawing proved to be the most important techniques as the analysis proceeded.

3. COMMUNICATION BARRIER TOWARDS CUSTOMERS

Generally speaking, the problem presented in this section is a demonstration of what happens when inexperienced computing students encounter occupational reality. The section elaborates the core category whereby inexperienced students' entry into a realistic software development context discloses a communication barrier the students potentially have towards customers. Particularly, this problem relates to the project start-ups as the students are able to improve their communication towards customers during the projects.

How this core category, shortly labeled as "communication barrier disclosed", was discovered and selected (cf. selective coding according to Glaser)? It originates from the teachers' far reaching observations. For example, a teacher with a nine-project experience identified students' communication towards other project stakeholders and problems in interpersonal relationships as the most critical factors that can potentially ruin a project. Since the first author has a close relation (teacher in the course) to such observation data, the course evaluation statements the teachers write for each project were also shortly examined. This sampling was regarded as a conscious act of "taking account of subjectivity" while it is not clear that one can and even should avoid subjectivity with a strategic act when using grounded theory, see [21, p. 336]. The aim here was to investigate whether and how communication issues had been paid attention to.

First, ten statements assessing the projects in the high end of the grading scale were sampled. They covered a 6-year period (2001–2006) and six different teachers. In one of the ten statements, a teacher expected more communication towards other stakeholders. The rest described communica-

[2]A project has been characterized as a *temporary organization* [20].

tion with the terms good, active, and excellent, and often associated communication with the project progress. The statements assessing projects in the middle (six statements 2002-2008, seven teachers) and the low end (three statements 2001, 2002, 2004, one teacher) of the scale were also (randomly) sampled. Interestingly, the description of the communication performance followed the grading from the high end to the middle area of the scale. At the low end there had been serious intra-group problems in addition to or instead of communication problems towards other stakeholders. Based on this sampling, the relevance of students' communication towards other stakeholders was in line with the teachers' observations (word-of-mouth knowledge). The following data examples are from the teachers' evaluation statements.

[CT][3] From the high grade statements:
-Overall, the attitude to work was positive, which was demonstrated by unprompted communication towards teachers and customers...
-Communication towards customers and teachers was excellent during the whole project.

[CT] From the mid grade statements:
- Even though the group clearly had ideas, they did not actively bring them out for the attention of the whole project organization. The performance was improved during the project.
-The project tracking and team's outward communication faltered during the project, which caused some unawareness for the teacher and most probably for the team also.

[CT] From the low end:
-Regarding many of the solutions the team made, the team did not inform the teachers and the customers.

The above highlights students' communication towards other stakeholders, not particularly towards customers. Indeed, at the beginning of the projects, a communication barrier appears not only towards customers. However, the students are better able to communicate within their group and with the teachers. The teachers can take initiatives, thus gently force the students into dialog. Also, majority of the students are at least somehow used to communicating with other students. This is basically why this study focuses on students' communication towards customers. Also, as the first author has used "teaching as coaching" during the last three years, hence able to closely monitor the students' performance, further confidence was obtained concerning the relevance of this communication direction.

This problem (core category) has thus been identified in the course and it is relevant for those involved (teachers, students, and customers) — because it has consequences. This communication barrier means that a necessary feedback loop between the students and the customers does not emerge and misunderstandings appear. As a consequence, the students develop wrong software and a project may considerably slow down. The problem affects the project outcome and decreases students' satisfaction in their own performance, as illustrated with the following student experience.

[CS] *At the end, a doubt emerged whether we had interpreted the priorities correctly. The project focused on telecommunication and encryption modules, but was the customer actually more interested in database modules? Too often we made decision by ourselves and did not utilize the teachers' and the customers' opinions and expertize.*

[3]In the data examples, the marking CT refers to an example from the teacher data and CS to an example from the student data.

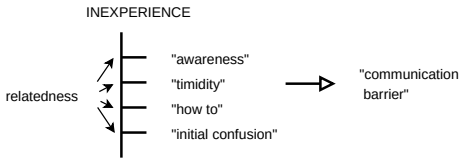


Figure 3: How inexperience creates the barrier

3.1 Inexperience

Students' inexperience was discovered to be the common explanation for the barrier. In general, the students express their inexperience as they tell about the doubts they had before the course concerning their skills. Many of them also explicitly tell that they did not have earlier software project experiences. This section explicates the ways how the students' inexperience creates the barrier. These "ways" are referred to as dimensions of inexperience, and are depicted in Figure 3.

First, the students are not aware of the importance of communication with a customer (*awareness*). They are not sufficiently aware of this kind of key areas in software development work. They just don't know they have a problem. This problem of awareness is evident as the students are better able to reflect on the project areas which are "close" to their course work, including group work and technical learning. In the students' course experiences, such "close issues" have more volume compared to the issue of communicating with the customer. This problem of awareness is also illustrated as there is a difference in the students' and the teachers' preferences. Whereas the students focus on the "close issues" other than customer communication, the teachers have regarded communication as a critical success factor in the projects. The following example illustrates the problem of awareness.

[CS] *It was not always clear, at least to me, what the customer wanted. When discussing resourcing questions [resource module in the software] I should have listened better in the first meetings, as I did not expect that I'm the one who had to deal with it [implement it]. Fortunately, I got answers to important questions later.*

Second, the students are timid to communicate with the customers (*timidity*). As they lack project experiences they are unconfident when starting to collaborate with the customer. They, for example, seem timid to communicate uncertain topics and remain silent in the situations where the teachers notice that the topic discussed with the customer is far from clear at that point. This dimension is a sensitive issue for the students. For example, whereas a teacher observes timidity, the students demonstrating it do not mention such issue in their course feedback. The following data examples illustrate the students' timidity.

[CS] *I could have been a bit more active and trust my skills more. Towards the end, I became more active and I dared to state my opinions more.*

[CT] *A teacher observation comes from the case were a student group was made aware of the importance of communication and encouraged to communicate. The group, however, demonstrated considerable timidity to communicate*

with the customer and started to talk to teachers (the supervisor in charge and the student supervisor) in the meetings while it should have talked to the customer sitting in the same table. It then happened that also the customer started to communicate with the students through the teachers. The team was not able to make progress when the teacher encouraged the team outside the meetings. The solution was that the teachers came early to the meetings and took seats so that the customer and the student group had to always sit opposite to each other. In addition, the teachers started to channel the communication out of themselves with small polite expressions. These helped. The students did not know about this effort. The group started to develop autonomy in communication with the customer.
Comment: this case illustrates the sensitivity of the timidity dimension.

Third, the students do not know how to communicate (*how-to*). They do not have means to tackle a new problem domain, which slows down the project and leads to misunderstandings. The students do not know, for example, that concretization (using drawings, managing open questions, and drafting user interfaces) would help their communication with the customer. At the end of the project, in their course feedback, they have sometimes a blaming tone concerning customer's activity while they do not see their own part in the problem, i.e., that they can and should take initiatives. In short, they are not aware of the means nor the roles or responsibilities when communicating with a customer in a variety of situations. The following two examples illustrate the how-to dimension.

[CS] *What proved to be problematic, was that, at times, the customer did not understand the challenge of the requirements from a technical point of view. Communication with the customer was surprisingly difficult.*
Comment: Here the student does not know how to manage a collision of technical and non-technical world when communicating with the customer. From another perspective, how to manage customer's expectations.

[CS] *On the other hand, a very difficult issue in the project was that three computer science students tried to figure out what kind of system the library needs for book supply.*
Comment: This illustrates the unawareness of the importance of communicating with the customer, but also the challenge the new domains create for the students (how to communicate to efficiently learn the domain).

Fourth, the inexperienced students suffer from initial confusion at the beginning of the projects, which is due to new people, new problem domain, inexperience in software processes and project management etc. (*initial confusion*). This complicates the students' chances to focus on single key areas of project work, such as communication with a customer. This is thus a specific dimension that indirectly has effect on the students' communication with the customers. A student comment that explicitly brings this up is found.

[CS] *Communication within the group and with other stakeholders was difficult at the beginning but was considerably improved as the project drew to a close. I suppose this happened because the stakeholders knew each other better, and we got rid or the general confusion of the start-up.*

The above dimensions of students' inexperience are related with each other (*relatedness*). The fourth item is actually one explanation for the first item: initial confusion makes it difficult for the students to notice the importance of communication. Another identified form of relatedness is demonstrated when insufficient knowledge on how to communicate, for example, not knowing what is the appropriate style in communication, appears as timidity.

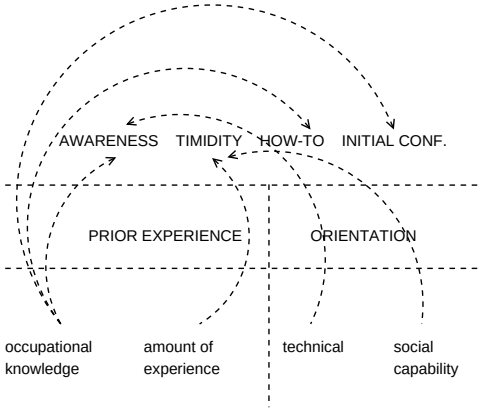


Figure 4: Individuality influenced by personal background factors

3.2 Individuality

The communication barrier is individual to each student. Which dimensions of inexperience a student demonstrates, and how strongly the dimensions appear, depending on the student's personal background factors, defines how the communication barrier appears in the student's performance.

Thus, the dimensions of inexperience the students demonstrate vary. For example, within a single group, some of the students point to their timidity whereas others to the problem of awareness. Second, there is also lots of variance within a single dimension, for example, in that of how timid a student is to communicate with the customer. Some students are first able to communicate only when it's their duty, for example, when it's their turn to be the chairman of the meeting. On the other hand, one student started to actively communicate with the customer before the first meeting, without any supervision, hence taking a leadership in communicating with the customer right away.

This individuality is explained by students' personal background factors: *personal orientation* and *prior experience*. Figure 4 depicts the relationships that will be explained in the following.

Consider first the personal orientation of a student. Some of the students demonstrate a strong personal interest in technical issues. This hinders the students from seeing how important it would be to actively communicate with the customer. Another aspect to personal orientation is social capability having effect on whether and how strongly timidity appears.

The effect of prior experience is illustrated with the finding that a socially capable student, the active student referred to above, knew that the group work should be quickly organized. Here, the good performance was not only based on social capability but also to some kind of understanding of what would hinder the team's progress (here the "what" was initial confusion causing unorganized thus inefficient work). Thus, as is obvious, also students' experience background

explains the individuality. The experience background encompasses two viewpoints here. The first viewpoint is the experience in the form of prior occupational knowledge that mitigates the problems of awareness, how-to, and initial confusion. The second is the amount of experience, i.e., whether the students have previously been exposed to customer projects at all having effect on how timidity appears.

3.3 Progress

The students demonstrate the barrier at the beginning of the projects but are able to make progress during the project. The term "progress" thus means here that the students start getting rid of the barrier. The progress takes place due to 1) course arrangements, 2) general progress of the project, and 3) instruction. In the following, these three properties are discussed in relation to the dimensions of inexperience.

Awareness. Course arrangements increase the students' awareness on customer communication in that the students are exposed to real life problems and real customer expectations. As the students struggle with the new problem domain, and see the consequences of misunderstandings on the domain knowledge they become better aware of the importance of customer communication. Additional challenge in the arrangements further increases the students' awareness. For example, when the problem domain is considerably challenging or the customer is from another city (geographically long distance between the university and customer), the importance of communication with the customer is better noticed. As the students make general progress in how to manage their work, i.e., they start to organize their work and get rid of initial confusion, they are better able to notice the importance of this kind of key areas of a customer project. The students need to be repeatedly instructed to help those that are technically oriented, and, in general, to change/extend the students' prevailing conceptions of the contents of software development work.

Timidity. The course arrangements decrease timidity as each group meets their customer regularly and is hence exposed to situations that require communication with the customer. The students are expected (not forced by instruction) to share tasks that require customer communication, meaning that each student is exposed to customer communication. They get to know the customer and get rid of initial confusion caused by the work with many new people. General progress of a project (contributed by the course arrangements) also reduces timidity. As the students notice that they are able to advance the project, they become more confident and their communication with the customer is improved. Instruction reduces timidity, but as the timidity is a very sensitive subject for the students, it must also be handled in a sensitive manner. The students must be patiently and repeatedly supported. Actually, the progress regarding timidity can be accelerated only to a certain point[4].

How-to. The students do not consider concrete "how-to" guidelines in their course feedback. Some students demonstrate how-to skills but by comparing such observation with the students' feedback not including any explicit reflection on this, it seems that the students have tacit implicit knowledge due to prior experience. Without help they probably learn something regarding the how-to dimension, but it remains implicit. The students can be instructed how to communicate and given concrete practices, and this results in improved performance. However, because the dimensions of the students' inexperience are related to each other, knowing how to communicate does not lead to a straightforward success. For example, insufficient awareness on the importance of communication has to be managed to make use of "how-to" -knowledge. Some students do not make use of communication tips as they do not see the importance of communication. This form of relatedness was thus found by observing the students' progress from the instruction viewpoint.

[4]Even though a teacher asks the "stupid questions" from the customer to mitigate the tension and to get the students involved (cf. peripheral participation [15]), the progress pertaining to timidity remains slow with some students. This may be a cultural issue. Gibbs [7] has noted that students need to be provided with the space where they can process a problem with their own language. This may also explain the slow progress.

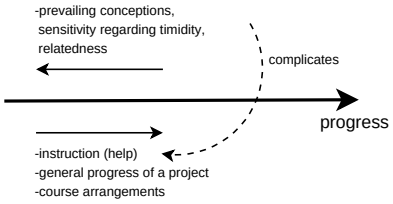


Figure 5: Students' progress

Initial confusion. Both the student data and the teachers' observations indicate that a slow project start-up due to the initial confusion is a frequently identified problem in the course. The students are able to organize their work if they are coached to adopt a process (roles, practices etc.). Course arrangements also contribute to the adoption of a process as the students are, for example, expected to plan a project (write a project plan).

In short, the progress is often slow without help. Some students notice the communication issues late, at the time of their course feedback, and it seems that the slow performance decreases their satisfaction on their own project performance. The students' performance can be enhanced by instruction which does not however lead to a straightforward success. The difficulty of instruction is explained by the students' prevailing conceptions, sensitivity regarding timidity, and relatedness of the dimensions of inexperience — as discussed above. The students' progress is illustrated in Figure 5.

4. ILLUSTRATING THE USE OF GT

DGT [10, pp. 101–115] introduced a constant comparative method for qualitative analysis which was followed in this research. The method encompasses four overlapping stages which are 1) the comparing incidents applicable to each category, 2) integrating categories and their properties, 3) delimiting the theory, and 4) writing theory.

The basic idea of comparison (stage 1) is that, as coding an incident, the analyst compares the incident with the previous incidents coded in the same category. This takes place both within the same and different comparison groups. The second rule of comparison is that analyst should at times stop coding and record a memo on his/her ideas. Integration

(stage 2) means the discovery of how concepts and their subconcepts interrelate, and hence, result in a unified whole. As the coding proceeds, the analyst compares, not only incidents with some other incidents, but with the properties of the category that resulted from initial comparison of incidents. This leads to integration. The constant comparison is said to naturally lead to the discovery of integration.

The comparison and integration are illustrated in the following. As the communication barrier was studied, the first explanation was that the students do not sufficiently know the importance of communicating with the customer. Then, when analyzing another incident where the student group seemed to have problems in communicating with the customer, it was noticed that this group was made aware of the importance of communicating, but they still did not communicate sufficiently. Here the incident was compared with previous incident coded in the same (main) category and this disclosed that there has to be another explanation. The property discovered was the students' timidity which also seemed to significantly explain the students' difficulties. Codes on the students' inexperience in general were also found. In these the students brought out that they did not have project experience prior the course, and spoke of their doubts they had prior to the project concerning their skills to succeed in the project. A hypothesis was generated that timidity is probably at least partly due to the lack of experiences in customer projects.

Then, as the analysis proceeded, it was discovered that some students who had difficulties in noticing the importance of customer communication, demonstrated a clear technical orientation. These students keep focusing on design and implementation, not noticing that communicating with the customer is a necessity to get feedback on the project outcomes. Integration was here very straightforward: the individual background factor explains the students' difficulty to notice the importance of communicating with the customer.

According to DGT, delimiting the theory (stage 3) is forced with the constant comparison. Theory becomes solidified, modifications become fewer and the emphasis turns to clarification and reduction. DGT says that the analyst starts to achieve two major requirements of a theory: parsimony of formulations needed and applicability to a wide range of situations, while keeping the theory close to data. In this study, delimiting of the theory started from writing notes. The second rule of comparison (stop coding and write notes) was thus important from the delimiting perspective. Glaser later emphasizes the thinking process needed and writing of memos [8, pp. 7, 83].

Delimiting the theory is illustrated in the following. An incomplete theoretical framework that emerged form the student data is depicted in Figure 6. It originates from the point where the coding was terminated and time was spent on sketching the relations between the concepts. The drawing is a note copied exactly as was handwritten in the coding booklet. The important issue here is reduction. The figure presents yet an unclear view of the findings as a whole. As the analysis proceeded, an overlap of the properties in the figure's framework was noticed, and reduction could be made. It was noticed that actually both timidity and unawareness of importance of communicating can be explained in terms of inexperience. The outline of the conceptualization being generated could hence be stated more clearly and

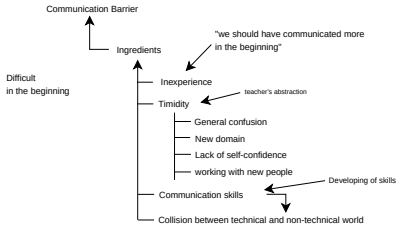


Figure 6: An incomplete theoretical framework from the analysis on the student data

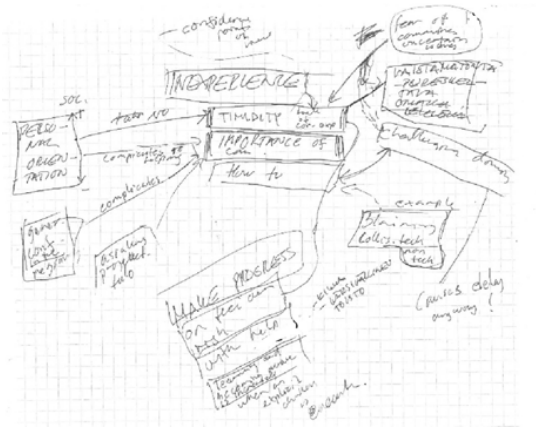


Figure 7: A draft of overall conceptualization

with fewer main categories. In this figure, the problem of awareness is only illustrated by a student cite "we should have communicated more in the beginning".

Figure 7 illustrates another attempt to make reduction and explicate the logic of the findings. The contents of the drawing are very close to what was finally the output of this research thread. Here, the inexperience of the students is a central category to which the other categories and their properties seem to interrelate. The category of how students make progress regarding the barrier has now been identified. The third category, that of how this problem (barrier) is individual to each student, is emerging as the figure includes notes on students' personal orientation. These two drawings are just a few of the attempts to make reduction and explicate the findings during the research. Again, the second rule of the comparison is in the key position: lots of time is needed to make sense of the data in order to discover integration and delimit the theory being generated.

In DGT, writing of a grounded theory (stage 4) is suggested to be started from the core category that puts forward the theory. This is followed by going through the main themes of the theory. Glaser points out that when writing an outline of the theory, the analyst may notice that integration falls apart. He suggests that the writing should be started anyway, as it potentially leads to reintegration of what has fallen apart [8, p. 132]. Writing from this methodical perspective was considered very useful in this work. The following cite well characterizes the challenge that was encountered [8, pp. 128–129]: *Rather, writing must capture it.*

It must put into relief the conceptual work and its integration into a theoretical explanation. So very often in qualitative research, the theory is left implicit in the write-up, as the analyst gets caught up in the richness of the data.

Indeed, the writing was about reduction and explication, about making implicit knowledge sufficiently explicit. Drawings were used in parallel with the writing. Figure 7 was actually sketched to support writing. The writing forced to think of sufficiently explicit conceptualization at a high level of abstraction. Writing of small textual fragments and outlining the unified whole was very difficult. The difficulty was actually surprising as it sometimes took hours to formulate a fragment of the outcomes in a consistent textual form that carried the correct idea. These formulations had yet to be reworked several times. This was not deduction and forcing but reduction and explication.

5. DISCUSSION

Based on an initial literature comparison, it seems that results in line with this study exists. If one considers the considerable realism provided by the project course at JYU/-MIT, a comparison can be made to experiences of employed graduates who have been found to encounter a variety of communication challenges after employment. For example, in [18], many data extracts reflect the variety of working life situations where one needs to know *how to* communicate. This is in line with the third dimension of inexperience found in this study and suggests that one possible way to further the how-to dimension would be to recognize and characterize the variety of project situations involving customer communication. On the other hand, one can consider Williams's et al. [26] results which indicate that students' performance depends on how students develop confidence. This is in line with the finding that students are better able to communicate with customers as they become more confident, after noticing some progress in their course work.

The study indicates the problem that all the students are not able to identify the variety of software engineering key areas without help. Consequently, communication should be an issue already in the early curriculum to provide students with relevant expectations of the contents of software engineering work. Lethbridge et al. discuss the wrong beliefs and expectations young people may have on software engineering, hindering their entry to the field [16]. Obviously, the beliefs may live during the course of university studies if the curriculum does not provide the students with proper learning contexts so that the issues other than technical ones are not given sufficiently attention. The results of this study indicate that changing students' conceptions through instruction is likely a slow process.

Importantly, this study illustrates how teachers gain knowledge and explicate their existing tacit knowledge through conceptualization. The dimensions of the students' inexperience provide a sufficiently easy-to-remember framework by which teaching practices can be designed and evaluated. In line with Glaser [8, p. 14], a teacher possessing a relevant substantive theory, can "work with familiar occasions purposefully".

5.1 Use of Grounded Theory

In this work, DGT and Glaser's approach on GT was followed. Many others have taken the approach of Strauss and Corbin [24]. For example, Coleman and O'Connor [4] followed Strauss and Corbin and their study differs from this study in several ways. In their study, it is difficult to identify such a core category that would provide a point of reference to the results. The other difference is that the study first develops hypotheses and then tests them, and one must here notice that Strauss and Corbin suggest that the analysis is a continuous interplay between inductive and deductive thinking [24, p. 111]. The hypotheses are deduced and then verified.

In computing education studies using GT, the research process has often been pre-designed and the use of GT is about data analysis in the process of *collect data → analyze main themes*, and sometimes *→ verify the themes with the data*, e.g. [12]. Also, authors report partial results from on-going studies [13], which indicates that GT is time consuming. GT is taken as a set of analysis techniques in case studies [5] and it is referred to when there is an inductive part in the research process [3]. There seems to be variance in the method usage which motivates further experimenting with GT in the context of computing education research.

Grounded theory process was not perceived as a mechanical process, and this made it difficult to explicate the research schedule and data usage in Section 2.2.1. In line with this, Piantanida et al. [21, p. 341] concluded that a traditional scientific report that is expected to reflect procedural rigor is atheoretical and problematic for a grounded theory. Another viewpoint is, as noted in [1], that the mechanical approach tends to result in a product of content analysis, not a theory. Too much focus on rigorous coding in a technical sense may hinder the analysis thus not allowing theoretical sensitivity. From a more practical viewpoint, explication and reduction of the emerging overall conceptualization were considered the most difficult tasks of the study. The concepts were found as well their partial integration. But to be able to see the rough skeleton, the essence of the conceptualization, required serious efforts. Writing of consistent textual formulations accompanied with drawings was experienced very helpful in this effort.

The data of the work was relatively small[5], but it enabled carrying out a GT process and evaluating the method usage. Using both the students' experiences and the teachers' observations was necessary here. While the communication issues had not much volume in the student data compared to other issues (e.g. group work), it was considered one of the most central success factors among the teachers. This was possible to notice using both the data. Furthermore, without the student data, all of the viewpoints would not have been included in the conceptualization, for example, how initial confusion complicates the students' chances to become aware of issues such as communicating with a customer. Here the point is that the systematic method using both teachers' observations and the perceptions of actual study subjects helps to identify all the relevant aspects even though they would already exist as tacit knowledge, in an implicit form. As Glaser notes, we (observers) do not know better than they (observees) [9, p. 49]. Altogether, using GT in this kind of educational study, which aims to know what is going on in students' performance, having also relation to teaching, necessitates use of data from both students and teachers.

[5]Notice the "teacher as a researcher" setting which implies that the researcher is surrounded by the data.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In this article, students' communication with customers at a capstone project course context was studied using Grounded Theory. A framework manifesting students' inexperience was found, the framework to which the students' individual factors and the progress they make, were found to interrelate. The study yielded explicit knowledge useful for teaching, and from this perspective, the chosen research method suited the research interest. Based on this experience, it is argued that conceptualizations are needed in educational research, and in this effort, grounded theory provides a means to bring out sufficiently analytic knowledge that has real value for improving teaching. A general implication to teaching is that communication skills should be gradually developed throughout the curriculum in order to develop students' confidence and awareness on the communication issues.

The work gives rise to a number of future directions. The found dimensions of inexperience could be studied in the context of a whole curriculum and furthered by a systematic comparison to literature (literature as data). An interesting question would also be that what kind of project issues prompt students' awareness of customer communication. The results of this paper indicate that projects starting from the scratch with a new problem domain for the students might have a best value in this respect. In order to develop a more formal theory [10, pp. 32–33], the results of the work provide a starting point to study a potential existence of the communication barrier in other educational contexts, where the occupational reality is first encountered. A more formal theory would be achieved by comparing the results to other fields, for example, the fields with an established culture regarding the importance of communication in a service provider–customer -relationship.

7. REFERENCES

[1] S. Adolph, W. Hall, and P. Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 166–178, New York, NY, USA, 2008. ACM.

[2] A. Berglund. What is good teaching of computer networks? *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, 3:S2D–13–18 vol.3, Nov. 2003.

[3] D. Chinn, C. Spencer, and K. Martin. Problem solving and student performance in data structures and algorithms. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 241–245, New York, NY, USA, 2007. ACM.

[4] G. Coleman and R. O'Connor. Investigating software process in practice: A grounded theory perspective. *Journal of Systems and Software*, 81(5):772 – 784, 2008.

[5] K. Deibel. Studying our inclusive practices: Course experiences of students with disabilities. *SIGCSE Bull.*, 39(3):266–270, 2007.

[6] P. Freeman, A. I. Wasserman, and R. E. Fairley. Essential elements of software engineering education. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 116–122, Los Alamitos, CA, USA, 1976. IEEE Computer Society.

[7] G. Gibbs. *Teaching students to learn: A Student-Centred Approach*. The Open University Press, Milton Keynes, England, 1981.

[8] B. G. Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, San Francisco, CA, 1978.

[9] B. G. Glaser. *Emergence vs. Forcing: Basics of Grounded Theory Analysis*. Sociology Press, Mill Valley, USA, 1992.

[10] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York, 1967.

[11] B. H. Hansen and K. Kautz. Grounded theory applied - studying information systems development methodologies in practice. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 8*, page 264.2, Washington, DC, USA, 2005. IEEE Computer Society.

[12] M. Hewner and M. Guzdial. Attitudes about computing in postsecondary graduates. In *ICER '08: Proceeding of the fourth international workshop on Computing education research*, pages 71–78, New York, NY, USA, 2008. ACM.

[13] C. Ho, K. Slaten, L. Williams, and S. Berenson. Work in progress-unexpected student outcome from collaborative agile software development practices and paired programming in a software engineering course. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages F2C–15–16 Vol. 2, Oct. 2004.

[14] S. Hornik, H.-G. Chen, G. Klein, and J. Jiang. Communication skills of IS providers: an expectation gap analysis from three stakeholder perspectives. *Professional Communication, IEEE Transactions on*, 46(1):17–34, Mar. 2003.

[15] J. Lave and E. Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, New York, 1991.

[16] T. C. Lethbridge, J. Diaz-Herrera, R. J. J. LeBlanc, and J. B. Thompson. Improving software practice through education: Challenges and future trends. *Future of Software Engineering, 2007. FOSE '07*, pages 12–28, May 2007.

[17] U. Melin and S. Cronholm. Project oriented student work: Learning & examination. *SIGCSE Bull.*, 36(3):87–91, 2004.

[18] S. Nagarajan and J. Edwards. Towards understanding the non-technical work experiences of recent australian information technology graduates. In *ACE '08: Proceedings of the tenth conference on Australasian computing education*, pages 103–112, Darlinghurst, Australia, Australia, 2008. Australian Computer Society.

[19] J. Norback and J. Hardin. Integrating workforce communication into senior design. *Professional Communication, IEEE Transactions on*, 48(4):413–426, Dec. 2005.

[20] J. Packendorff. Inquiring into the temporary

organization: New directions for project management research. *Scandinavian Journal of Management*, 11(4):319 – 333, 1995.

[21] M. Piantanida, C. A. Tananis, and R. E. Grubs. Generating grounded theory of/for educational practice: The journey of three epistemorphs. *International Journal of Qualitative Studies in Education*, 17(3):325–346, 2004.

[22] J. Polack-Wahl. It is time to stand up and communicate [computer science courses]. *Frontiers in Education Conference, 2000. FIE 2000. 30th Annual*, 1:F1G/16–F1G/21 vol.1, 2000.

[23] I. C. S. Press and A. Press. IEEE/ACM joint task force on computing curricula. Software Engineering 2004, curriculun guidelines for undergraduate degree programs in software engineering. Retrieved May, 2007, from http://sites.computer.org/ccse/, 2004.

[24] A. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Newbury Park, California, 1990.

[25] G. Taran. Managing technical people: Creatively teaching the skills of human interaction in today's diverse classrooms. *Software Engineering Education and Training, 2008. CSEET '08. IEEE 21st Conference on*, pages 93–100, April 2008.

[26] L. Williams, L. Layman, K. Slaten, S. Berenson, and C. Seaman. On the impact of a collaborative pedagogy on african american millennial students in software engineering. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 677–687, May 2007.

Recalling Programming Competence

Jens Bennedsen
Engineering College of Aarhus
Dalgas Avenue 2
DK-8000 Aarhus C, Denmark
jbb@iha.dk

Michael E. Caspersen
Department of Computer Science
Aarhus University
DK-8000 Aarhus C, Denmark
mec@cs.au.dk

ABSTRACT

Programming is recognised as one of seven grand challenges in computing education and attracts much attention in computing education research. Most research in the area concerns teaching methods, educational technology, and student understanding/misconceptions. Typically, evaluation of learning outcome takes place during or immediately following the educational activity. In this research, we conduct a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place. Our population consists of ten students who have taken an introductory object-oriented programming course 3, 15, or 27 months prior to our study. None of the students have been exposed to programming in the intervening period. As expected, our research shows that syntactical issues in general hinder immediate programming productivity, but more interestingly it also indicate that a tiny retraining activity and simple guidelines is enough to recall programming competence and overcome syntactical issues.

Categories and Subject Descriptors

K3.2 [**Computers&Education**]: Computer and Information Science Education—*computer science education, information systems education*

General Terms

Experimentation, Human Factors

Keywords

CS1, object-oriented programming, remembering

1. INTRODUCTION

For many years there have been a massive interest in program education research and development. Teaching methods, materials, and educational technology have been developed to help students better learn how to program. Some of these innovations have been systematically evaluated for their impact, but in general the measurement of success is defined by how well the students perform at the final exam or at tests during the course. The quest for success indicators is an example of such studies (see e.g. [8, 42, 7]), and so are studies that evaluate educational technology (see e.g. [38, 24, 21]). Evaluating the impact immediately after the course, is of course both interesting and relevant, but in general the goals of our teaching is not only that the students perform well at the final exam, but that the students achieve relevant and lasting programming competences.

Computing competences are becoming relevant in many fields; consequently, many students who will not major in computer science will be required to take an introductory computing course [18]. Many introductory computing course has programming as a core activity and learning goal, and for good reasons since programmability is the defining characteristics of the (digital) computer. This is also echoed in the ACM/IEEE curriculum recommendations. Currently, a revision and enlargement of the curriculum recommendations is under way, broadening the scope from traditional computer science to the broader field of computing [35], from Information Systems [16] to Computer Engineering [37].In e.g. "the model curriculum and guidelines for graduate degree programs in information systems" [17] it is noted that *Students entering the MSIS program need the content of the following courses ... programming* (p.138).

We forget things. The cognitive structures that store facts and schemes typically become less accessible over time, and forgetting is more likely to take place when memory elements are not accessed and used [9]. By fitting data from several experiments in cognitive psychology, Woodworth [45] created the so-called forgetting curve, see figure 1. Accordingly, it should be expected that students do not have the same competences say one year after an exam as they had right after the exam.

In our current research, we are particularly interested in studying the durability of programming competences achieved in an introductory object-oriented programming course for non-CS majors. We have conducted a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place. Our population consists of ten students who have taken an introductory object-oriented programming course 3, 15, or 27 months previous to our test. None of the students, who are

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

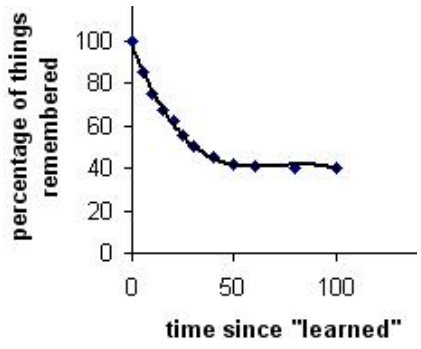


Figure 1: Classic shape of the forgetting curve (Woodworth, 1938).

majors in bio-technology, have been exposed to programming in the intervening period.

The remaining part of the article is organised as follows: Section two describes related work primarily in cognitive psychology. In section three and four we describe the instructional design of the introductory programming course. Section five presents our hypotheses and research questions, and section six presents our research design. In section seven we describe and analyse our observations. Potential future work is described in section eight, and section nine is the conclusion.

2. RELATED WORK

This section describes the related work. It focuses on two things, work in the area for forgetting and work in the area of remembering programming competences.

2.1 Memory

The memory is fallible. The fact that we gradually forgets was first documented by Ebbinghaus [14] in a study where he first tried to learn nonsense syllables and then tried to remember as much as possible at various delays after the learning. His conclusion was, that there was a rapid drop-off in retention in the beginning and then a more gradual drop-off later. As he wrote: *One hour after the end of the learning, the forgetting had already progressed so far that one half the amount of the original work had to be expended before the series could be reproduced again; after 8 hours the work to be made up amounted to two thirds of the first effort. Gradually, however, the process became slower so that even for rather long periods the additional loss could be ascertained only with difficulty. After 24 hours about one third was always remembered; after 6 days about one fourth, and after a whole month fully one fifth of the first work persisted in effect* (section 29, [15]). Ebbinghaus found that a complex logarithmic function described his data. Later it has been shown [43] that a power function $y = \alpha t^{\beta}$ better describes the relation between time and remembering. The values of α and β relies upon the actual person and the "thing" to remember.

Apart from an interest in forgetting, Ebbinghaus was also interested in the effect of repeated learning. He found that *The relation is quite similar to that described in Chapter VI [the relation between time and forgetting] as existing between the surety of the series and the number of its repetitions* (section 31, [15]).

Relearning affects forgetting. As Schacter [33] notice *it is known, for instance, that retrieving and rehearsing experiences play an important role in determining whether those experiences will be remembered or forgotten* (p. 184). The current memory model is actually more complex than a simple correlation between recall and remembering. Loftus [25] have found four major reasons why people forget: retrieval failure (memory traces decay over time), interference (memory may compete and interfere with other memory) , failure to store (e.g. details may be filtered out) and motivated forgetting (we want to forget e.g. traumatic things). As Anderson, Bjork and Bjork [2] notice *a striking implication of current memory theory is that the very act of remembering may cause forgetting. It is not that the remembered item itself becomes more susceptible to forgetting; in fact, recalling an item increases the likelihood that it will be recallable again at a later time. Rather, it is other items — items that are associated to the same cue or cues guiding retrieval — that may be put in greater jeopardy of being forgotten.* (p. 1063). According to Anderson, Bjork and Bjork the reason for this is three assumptions on how the memory work

the competition assumption Memories associated to a common cue compete for access to conscious recall when that cue is presented,

strength dependence assumption a cued recall of a memory will decrease as a function of increases in the strength of its competitors,

retrieval-based learning assumption Recall of a memory enhances subsequent recall of that memory.

The knowledge of forgetting have inspired many (primary) schools to evaluate their school calendar [13]. In general there seems to be an impact of a calendar model with many small breaks as opposed to one long summer break since students tend to perform better on tests with many small breaks rather than one large break. The effect of forgetting was notable particularly with respect to math facts and spelling. Findings in cognitive psychology suggest that without practise, facts and procedural skills are most susceptible to forgetting [12]. The categories of facts and procedural skills most likely encompass the idiosyncrasy of programming language syntax and programming skills which is the focus of our research.

2.2 Learning to Program

Many approaches to introductory programming education have been proposed including a procedures early approach [29], a top-down approach [19, 30], a graphics approach [26]. Even within introductory object-oriented programming, many different approaches exist: objects early [1], interfaces early [34], GUIs early [44], concurrency early [31], events early [39], components early [20], etc.

All of these articles about introductory programming education describe different (groups of) people's approaches. However, many are in the "Marco Polo" style of reporting research in introductory programming [41]; or, to be more precise, they argue that a certain approach is better than others based on the assumption that certain learning outcomes should be promoted.

To properly evaluate the long-time learning effect of a program course, we must take as starting point the intended learning outcomes (ILO) of the course. Whether the ILO focus on special features of the programming language, the process of program development, or something else, has an impact on how we must test and recall programming competences. In the following section we will describe the ILO for the introductory object-oriented programming course taken by our population of students.

3. TEACHING PROGRAMING USING A MODEL-BASED APPROACH

In [22] three perspectives on the role of a programming language are described:

Instructing the computer The programming language is viewed as a high-level machine language. The focus is on aspects of program execution such as storage layout, control flow and persistence. In the following we refer to this perspective as coding.

Managing the program description The programming language is used for an overview and understanding of the entire program. The focus is on aspects such as visibility, encapsulation, modularity, separate compilation.

Conceptual modelling The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena.

When designing a programming course, one must balance the three perspectives; in a model-based programming course, by definition, conceptual modelling plays the most important role. In the course under consideration, the progression in the course is defined not by the syntactical structure of the programming language, as is usually the case [32], but by the complexity of specification models, i.e. class models and functional specifications of methods. Early in the course, examples, exercises and assignments address programming tasks described by simple specification models (one class only or two classes with a simple relationship and simple functional specifications); later in the course the programming activities are defined by more complex specification models (more classes with more advanced relations and more complex functional specifications).

The official ILO for the course is phrased as follows: After the course, the students must be able to apply fundamental constructs of a common programming language, identify and explain the architecture of simple programs, identify and explain the semantics of simple specification models, implement simple specification models in a common programming language, and apply standard classes for implementation tasks.

The evaluation of programming competences in this course is done by a 30 minute practical exam. For a description of how we measure the students‘ programming competences, see [6].

For a more detailed description of the model-based programming course design, see e.g. [4, 10, 5].

4. THE PROGRAMMING COURSE

The programming course under consideration spans the first half of CS1 at Aarhus University. The course runs for seven weeks, and after the course there is a practical lab examination with a binary pass/fail grading. The grading is based solely upon the behaviour in and result of the final examination; acceptable performance in weekly mandatory assignments during the course is a prerequisite for the final exam but does not count as part of the grading. There are approximately 350 students per year from a variety of study programmes, e.g. bio-technology, chemistry, computer science, mathematics, geology, nano science, economy, and multimedia. 40% of the students are majors in computer science; of course they continue with many more programming or programming related courses. For most of the remaining students, this is the only mandatory programming course in their curriculum, but some choose follow up courses as electives and some do have special follow up courses related to their field (e.g. multimedia programming or scientific computing).

Content
Getting started: Overview of fundamental concepts. Learning the IDE and other tools.
Learning the basics: Class, object, state, behaviour, control structures.
Conceptual framework and coding patterns: Control structures, data structures (collections), class relationship, patterns for implementing structure (class relationship)
Programming method: Stepwise improvement, schemes for implementing functionality.
Subject specific assignment: Practise on harder problems.
Practise: achieve routine in solving standard tasks.

Table 1: Course phases

The students are grouped in classes of approximately 20 students; typically there are 17-18 teams per year. Each class has its own teaching assistant (TA) who is typically a PhD student in computer science.

We adopt an incremental approach to programming education in which novices are provided with worked examples [40] and initially do very simple tasks and then gradually do more and more complex tasks, including design-in-the-small by adding new classes and methods to an already existing design. Table 1 gives an overview of the phases and content of the course.

For a more detailed discussion of the design of the course from a learning theoretic perspective, see [11].

5. RESEARCH QUESTIONS

As described in section 2, we forget things, and forgetting is more likely to take place when memory elements are not accessed and used. Programming fluency involves a lot of specific skills related to the programming language (syntax, semantics, and pragmatics), the development environment (editor, compiler, interpretation of error messages, and debugging), use of API, etc. The first category of skills, which we denote concrete programming competences, implies that programmers possess a great deal of fingertip knowledge about many specific, technical details and is therefore particular vulnerable with respect to being forgotten when not practised and applied,. Another category of programming skills and competences relate to problem solving and appli-

cation of patterns to solve recurring (types of) problems; we denote this abstract programming competences. The examination form ensures that these programing skills and competences have been present, but how long and how well do they last, and how easy is it to recall them? Our two hypotheses, which forms the basis for this research, are:

Forgetting The students have forgotten the concrete programming competences quickly after they have passed the course.

Learning It does not take much effort for the students to recall the concrete as well as more abstract programming competences.

The two hypotheses are operationalised into the following research questions:

RQ_1: Forgetting Have the students forgotten their concrete programming competences?

RQ_2: Learning Can the students with a limited effort recall their programming competences? And what are the challenges for recalling once learnt skills and competences?

6. RESEARCH DESIGN

This section describes the design of the research.

6.1 Participants

From the general cognitive theory, we expect that the students' programming competences are forgotten if not practised and applied. Thus, in order to test our hypotheses and answer our research questions, we need to identify a group of students who have not programmed since they passed the introductory programming course. This naturally rules out computer science students. As described in the introduction, many other students take programming classes, but this is not the case for students majoring in bio-technology.

Students from bio-technology take the introductory programming course in the third quarter of the first year. They have no other mandatory programming courses, and they do not practise programming as part of their studies. These students fulfil the overall requirement (they have not been programming for X months) and they are a group that can be addressed, since most of them still follow the same study program. There are currently 45 students in the bachelor program of bio-technology (14 in the first year, 17 in the second year, and 14 in the third year). This makes it difficult to do quantitative analyses (the number of students are too small in each group). Consequently, we have designed the research not with the focus of giving general, generalisable answers but rather as providing new insight and pointers to factors it might be interesting to investigate further.

Based on the research questions and the group of students that are accessible, we observe the students performing programming with a focus on the problems they encounter as they go along. We do this twice: A pre-test before the students get a chance to brush-up of their programming competences, and a post-test after the students have received a brush-up. Finally we interview the students in a semi-structured focus group interview.

Year	Months since prog. course	Male	Female	Programming since course
2007	27	1	0	course using MathLab
2008	15	0	4	none
2009	3	2	3	none

Table 2: The students participating in the experiment

6.2 Evaluation of Programming Competences

A key question is how we can evaluate the students programming competences? The exam of the course evaluates the learning goals of the course and consequently the programming competences the students should possess. We evaluate the students using two programming tests similar to the one used in the final exam of the introductory programming course. In [6] we argue that the exam actually measures the goals of the course. The pre-test can be seen in the appendix; the post-test is similar to the pre-test except for another cover story.

6.3 Rehearsing Programming Competences

The next question is what "limited effort" mean (RQ_2)? Shall the try to recall the students' programming competences through practise or through a general presentation of key concepts, techniques, and examples? And shall we provide some kind of assistance to recall their programming competences during the post-test?

Ideally we would like to "measure" the learning effort it takes a given student to be able to solve the task in the pre- and post-test. This is in practice impossible! As a compromise, we offer the students an overview of the central programming language constructs (basic statements, control structures, method, attribute, class, etc.) and central concepts such as association (one-to-many) and collections and how these are realised in the programming language (Java). Furthermore, we give the students one of two kinds of help when solving the post-test. In the final focus group interview we specifically address how the learning aids have helped the students.

6.4 Concrete experiment design

We invited all bio-technology students from the first, second, and third year to participate in the experiment (45 in total). 12 responded positively to our invitation, and 10 actually participated in the experiment. The students were not paid (apart from a dinner at the end), nor did they get course-credit for the experiment. The students had the characteristics described in table 2.

The experiment was conducted a late afternoon in a computer-lab (the same that was used for the lab-sessions during the course) and lasted 3 hours. The agenda for the experiment was as follows:

1. Welcome and introduction
2. Short repetition of use of the development environment (BlueJ [23])
3. Pre-test
4. Brush-up of programming competences

5. Post-test

6. Focus group interview

The welcome and introduction motivated the study and gave a general overview of the content of the afternoon. This part took 15 minutes.

The repetition of the development environment helped the students to remember how the IDE was designed and how to edit and compile programs. This was done via a few exercises the students had to solve — exercises from the textbook used when the students had the course [3]. This was done in order to have programming in focus, not the tool used for programming. The exercises included a small amount of actual programming (the students typed in some code that was provided, they did not develop the solution themselves). The students had therefore seen some Java code just before the pre-test. This part took 15–20 minutes (some students finished before others).

The pre-test was a standard assignment from a final exam. Four researchers observed the students (2-3 students per researcher). When the students got stuck, we noticed the problem and evaluated how the students tried to solve the problem. If the students had been stuck for a long period of time, we helped the students to move on and noted this help. The test lasted 30 minutes; same duration as the ordinary exam.

The brush-up of programming competences was done using some general slides from the introductory programming course. The slides describe general concepts (object, class, attribute, method, constructor, parameter, type, statement, selection, iteration, association and collection) and how these look in Java. The students could ask questions and discuss during the brush-up session. Nearly all of the students' questions were about specific details in Java. The students did not do practical programming during the brush-up session. This part of the experiment lasted one hour.

Also the post-test was a standard assignment from a final exam. In order to evaluate different aids, we divided the students into two groups: One group received a model solution for the pre-test, the other group received a general description of how to implement classes, associations and two algorithmic patterns (that typically occur in exam assignments): (1) in a collection of objects, find one that matches a given criteria, and (2) in a collection of objects, find all that matches a given criteria. The first help was very concrete; the second incorporated the idea of pattern-oriented instruction [28] which was emphasised in the ordinary course. As for the pre-test, four researchers observed the students and noted their difficulties. The post-test was also time-boxed to 30 minutes.

The focus group interview lasted 35 minutes and focused on the students difficulties, the difference between concrete programming competences and general competences, the effect of the intermediate learning task (the brush-up of programming competences), the influence of the aids provided, and general comments.

7. OBSERVATIONS AND ANALYSIS

This section describes and analyses the observations made during the experiment and the final focus group interview in order to answer the two research questions.

Month since prog. course	Last completed exercise	Problems
3	8	Did extremely well. Used `compareTo` instead of `equals` for checking if strings are equal.
3	6	Many problems with syntax like forgetting a method name.
3	5	Many syntactical problems.
3	none	Declared attributes in the constructor.
15	none	Methods without a signature. Confused about the value of a name-attribute and a reference to the given object.
15	none	Parameters for the values of the attributes in the `toString()` method.
27	none	Many syntactical problems. The `toString()` method was implemented by returning a string literal instead of values of variables.
3	4	Did fairly well. Wrote statements directly in the class without a surrounding method, but worked it out by himself.
15	3	Called a non-existing method (`gettoString()`).
15	none	Declared an attribute called `toString`. Declared attributes in the constructor.

Table 3: Each student's performance in the initial test

7.1 Forgetting

In this subsection we will look at RQ_1.

As expected, the concrete syntax was a major problem for almost all of the students. As one of the students noticed in the interview: *You quickly forget when to type a parenthesis or a semicolon - you can remember that it is important that they are put in the right place, but where that is ….* Another student expressed it the following way: *I had many problems in the first test. I could not remember how to write it — the class and the other stuff — I could remember that this class was a class and you can create objects from it, but in the code, I could not remember what to write and how to call. I could remember that you had to return something … but how it should be written and worked, I had totally forgotten .*

There was a difference between the students who took the course three month ago and the other students. All of the students had problems with the specific syntax, but the "younger" students (measured in time since they had the introductory course) had significantly less problems than the "older" students as can be seen from table 3. One of the

students (number 1) would actually have passed the test if it had been a real exam.

If we look more closely at the problems many students encountered in the pre-test, they include the following:

Attributes Many declared the attributes in the constructor and found it very difficult to initialise them.

Parameters Many found it difficult to declare parameters. It seemed like they had the idea of passing information through parameters but the concrete syntax was a problem.

Screen output vs. return value Many implemented the `toString()` method using a `System.out.println(...)`, and could not understand the error "missing return statement".

Programming process Many students gave up on a given question and left it unsolved even though it was required to solve the next question.

In general we conclude that the students had forgotten their specific programming competences. Only one student (who took the course three months ago) could solve more than very basic programming tasks. This student would have passed, had it been a real exam.

7.2 Learning

In this subsection we will look at RQ_2.

After the students had refreshed their programming competences, they performed significantly better as can be seen from table 4. If the post-test had been a real exam, seven of the ten students would have passed it!

In general, the aid that was provided helped the students. All of the students who had the model solution from the pre-test, performed well. In fact, they would all have passed had it been a real exam.

The students used the model solution in different ways. Some students started out on their own and just used the solution when they encountered a problem they could not solve by themselves. As one student said: *I did not use it for the first six questions ... there were something about ArrayList, how to write it, otherwise it was only in the end where you have to write a for-loop, I could not remember how to write that. I do understand the meaning and what it is, but I cannot remember how to write it.* Others used it more systematically: *I become a little stubborn when I get such one [a solution]. I want to do it by myself ... but I used it anyhow [for most of the test] because there were many things I could not remember .*

The performance of the students who got the general description was somewhat more diffuse. In general, they performed significantly better than in the pre-test, but not all would have passed had it been a real exam. Some students found it difficult to put the general solution to practice.

In general the *Refreshment of programming competences* phase in the experiment helped the students. As one student said *I think it helped me a lot — the PowerPoint show — because I had completely forgotten all. I actually think I had forgot that there should be a list if it wasn't told.*

In the *Refreshment of programming competences* phase, many students had good and in-depth questions using correct terminology for programming concepts. We see this as

Month since prog. course	Last completed exercise	type of help	Problems
3	9	G	None.
3	9	S	Forgot to include statements in { }.
3	9	G	None.
3	2	G	Wrote literals instead of identifiers in the parameter list of the constructor.
15	7	S	None.
15	8	S	None.
27	5	G	Forgot to import `java.util.*`.
3	9	S	None.
15	8	S	None.
15	4	G	Instead of type-identifier pairs in the parameter list, she wrote identifier-identifier pairs where the first identifier was the attribute and the second was the parameter.

Table 4: Each student's performance in the second test. S referees to a solution of the initial test, G to a general description of how to implement different structures.

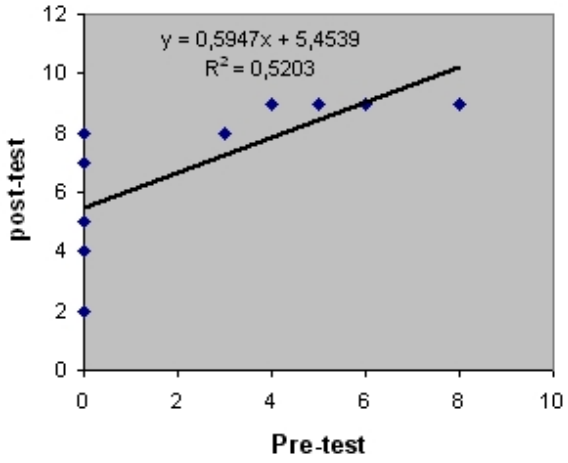


Figure 2: Number of completed exercises in the pre- and post test

an additional indicator that the students may have forgotten the syntax but the more conceptual content and general competences and skills are more easy to recall.

The design of this study was to use a qualitative research approach, where we observed what the students did, what problems they encountered and abstracted these findings. An alternative way to address the research question (RQ_2) could be to statistically check if the students performed better after the intervention. Figure 2 plots the students number of completed exercises in the pre- and post-test. If we analyse the data using linear regression [27], we can observe that there is a reasonably strog correlation between the observations ($R^2 = 0.52$), and that the line is well above the diagonal. This supports the conclusion that the intervention indeed helped the students recall their programming competences. However, as noted initially, the number of students in this study was only ten.

In general, we conclude that the students with the help they got (one hour of lecturing plus help during the test) could recall their programming competences. Consequently, we conclude that it is possible with a limited effort for most of the students in this study to recall general as well as more specific programming competences and skills.

The other part of RQ_2 "What are the challenges for recalling once learnt skills and competences?" is more difficult to answer.

8. FUTURE WORK

In this study only ten students from one study program participated. It will be interesting to expand the findings from this research by involving more students from more study programs. Fortunately, students from several other study programs who do not receive further programming instruction, have taken the course.

Programming is being taught in many different ways, and there are many different ways of phrasing the intended learning outcome of introductory programming courses. In order to obtain more reliable and generalisable results, it would be interesting to include more universities and colleges in the research and thus aim for a multi-institutional (and multi-national) study. As [36] argues, a *multi-national, multi-institutional context, defines a new interface between computer science education research and computer science education practice — hopefully bringing them closer together* (p. S4E-16).

9. CONCLUSION

We have conducted a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place.

In the pre-test, all students struggled with syntax issues, but the younger students (measured in time since they had the introductory course) had significantly less problems than the older students.

Our qualitative study indicates, not surprisingly, that syntactical issues in general hinder immediate programming productivity, but more interestingly it also indicate that a tiny retraining activity and simple guidelines is enough to recall general as well as more specific programming competences and overcome syntactical issues.

10. REFERENCES

[1] C. Alphonce and P. Ventura. Object orientation in cs1-cs2 by design. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 70–74. ACM Press, 2002.

[2] M. Anderson, R. Bjork, and E. Bjork. Remembering can cause forgetting: Retrieval dynamics in long-term memory. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 20(5):1063–1087, 1994.

[3] D. J. Barnes and M. Kï£¡lling. *Objects First With Java: A Practical Introduction Using Bluej*. Pearson, Essex, United Kingdom, 3rd edition, 2006.

[4] J. Bennedsen. *Teaching and learning introductory programming - a model-based approach*. PhD thesis, University of Oslo, Norway, department of Computer Science, 2008. accessed May, 2009.

[5] J. Bennedsen and M. Capersen. Model-driven programming. In J. Bennedsen, M. Caspersen, and M. Kï£¡lling, editors, *Reflections on the Teaching of Programming*, pages 116–129. Springer-Verlag, Berlin, Germany, 2008.

[6] J. Bennedsen and M. E. Caspersen. Assessing process and product ï£¡ a practical lab exam for an introductory programming course. *ITALICS, Innovation in Teaching and Learning in Information and Computer Sciences*, 6(4):183–202, 2007.

[7] J. Bennedsen and M. E. Caspersen. Optimists have more fun, but do they learn better? ï£¡ on the influence of emotional and social factors on learning introductory computer science. *Journal of Computer Science Education*, 18(1):1–16, 2008.

[8] A. J. Biamonte. Predicting success in programmer training. In *SIGCPR '64: Proceedings of the second SIGCPR conference on Computer personnel research*, pages 9–12, New York, NY, USA, 1964. ACM Press.

[9] R. Bjork. Retrieval practice and the maintenance of knowledge. In M. Gruneberg, P. Morris, and R. Sykes, editors, *Practical aspects of memory: Current research and issues*, volume 1, pages 396–401. Chichester, England, 1988.

[10] M. E. Caspersen. *Educating Novices in the Skills of Programming*. PhD thesis, Aarhus University, Department of Computer Science, 2007. accessed May 2009.

[11] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 111–122, New York, NY, USA, 2007. ACM.

[12] G. Cooper and J. Sweller. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79(4):347–362, 1987.

[13] H. Cooper, J. C. Valentine, K. Charlton, and A. Melson. The Effects of Modified School Calendars on Student Achievement and on School and Community Attitudes. *Review of Educational Research*, 73(1):1–52, 2003.

[14] H. Ebbinghaus. *ï£¡ber das Gedï£¡chtnis*. Teachers College, Columbia University, New York, New York, United States, 1885.

[15] H. Ebbinghaus. Memory: A contribution to experimental psychology. `http://psychclassics.yorku.ca/Ebbinghaus/index.htm`, 1885. Translated from German by Henry A. Ruger and Clara E. Bussenius (1913). Last accessed May 14, 2009.

[16] J. T. Gorgone, G. B. Davis, J. S. Valacich, H. Topi, D. L. Feinstein, and J. Herbert E. Longenecker. Is 2002 - model curriculum and guidelines for undergraduate degree programs in information systems. Retrieved June 2009, 2002.

[17] J. T. Gorgone, P. Gray, E. A. Stohr, J. S. Valacich, and R. T. Wigand. Msis 2006: model curriculum and guidelines for graduate degree programs in information systems. *SIGCSE Bull.*, 38(2):121–196, 2006.

[18] M. Guzdial and A. Forte. Design process for a non-majors computing course. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 361–365, New York, NY, USA, 2005. ACM.

[19] T. B. Hilburn. A top-down approach to teaching an introductory computer science course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, 25(1):58–62, 1993.

[20] E. Howe, M. Thornton, and B. W. Weide. Components-first approaches to cs1/cs2: principles and practice. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 291–295. ACM Press, 2004.

[21] J. Jain, I. James H. Cross, and D. Hendrix. Qualitative comparison of systems facilitating data structure visualization. In *ACM-SE 43: Proceedings of the fourty-third annual Southeast regional conference*, pages 309–314, Kennesaw, Georgia, 2005. ACM Press.

[22] J. L. Knudsen and O. L. Madsen. Teaching object-oriented programming is more than teaching object-oriented programming languages. In S. Gjessing and K. Nygaard, editors, *ECOOP '88 European Conference on Object-Oriented Programming*, pages 21–40, Berlin, Germany, August 15-17, 1988 1988. Springer-Verlag.

[23] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[24] R. B.-B. Levy, M. Ben-Ari, and P. A. Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40(1):1 – 15, 2003.

[25] E. Loftus. *Memory: surprising new insights into how we remember and why we forget*. Addison-Wesley, Reading, Massachusetts, United States, 1980.

[26] S. Matzko and T. A. Davis. Teaching cs1 with graphics and c. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 168–172, New York, NY, USA, 2006. ACM Press.

[27] D. C. Montgomery and E. A. Peck. *Introduction to linear regression analysis*. John Wiley, New York, NY, USA, 1982.

[28] O. Muller, D. Ginat, and B. Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.*, 39(3):151–155, 2007.

[29] R. E. Pattis. The ï£¡procedures earlyï£¡ approach in cs 1: a heresy. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 122–126. ACM Press, 1993.

[30] M. M. Reek. A top-down approach to teaching programming. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 6–9, New York, NY, USA, 1995. ACM Press.

[31] S. Reges. Conservatively radical java in cs1. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 85–89. ACM Press, 2000.

[32] A. Robins, J. Rountree, and N. Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[33] D. Schacter. The seven sins of memory: Insights from psychology and cognitive neuroscience. *American Psychologist*, 54:182–203, 1999.

[34] A. Schmolitzky. "objects first, interfaces next" or interfaces before inheritance. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 64–67. ACM Press, 2004.

[35] R. Shackelford, J. H. C. II, G. Davies, J. Impagliazzo, R. Kamali, R. LeBlanc, B. Lunt, A. McGettrick, R. Sloan, and H. Topi. The overview report. Accessed June 2009, 2006.

[36] B. Simon, R. Lister, and S. Fincher. Multi-institutional computer science educational research: A review of recent studies of novice

understanding. In *in Proceedings of the 36th Annual Frontiers in Education Conference*, pages SE412–17, October 2006.

[37] D. Soldan, J. L. Hughes, J. Impagliazzo, A. McGettrick, V. P. Nelson, P. K. Srimani, and M. D. Theys. Computer engineering 2004 - curriculum guidelines for undergraduate degree programs in computer engineering. Accessed June 2009, 2004.

[38] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 61–66, Amsterdam, The Netherlands, 1993. ACM.

[39] L. A. Stein. What we swept under the rug: Radically rethinking cs1. *Computer Science Education*, 8(2):118–129, 1998.

[40] J. Sweller and G. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985.

[41] D. W. Valentine. Cs educational research: a meta-analysis of sigcse technical symposium proceedings. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 255–259, New York, NY, USA, 2004. ACM.

[42] S. Wiedenbeck. Factors affecting the success of non-majors in learning to program. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 13–24, New York, NY, USA, 2005. ACM Press.

[43] J. Wixted and E. Ebbesen. Genuine power curves in forgetting: A quantitative analysis of individual subject forgetting functions. *Memory and Cognition*, 25:731–739, 1997.

[44] U. Wolz and E. Koffman. Interactivity in cs1 & cs2: bringing back the fun stuff with java. In *CCSC '00: Proceedings of the fifth Annual Consortium for Computing Sciences in Colleges CCSC: Northeastern conference*, pages 1–3. Consortium for Computing Sciences in Colleges, 2000.

[45] R. Woodworth. *Experimental Psychology*. Henry Holt, New York, United States, 1938.

Aarhus University
Dept of computer science

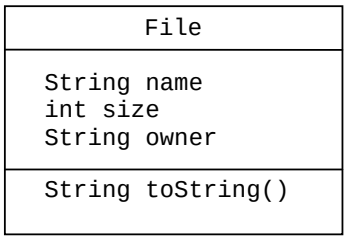
APPENDIX

Pre-test
June 3rd 2009

Experiment 1, (Pre-test) Post-test is similar except that another domain and slightly modified methods are used

1. Create a class, *File*, representing a file; the class *File* is specified in the UML-diagram to the right. The three attributes must be initialized in a constructor (using parameters of appropriate type). The method *toString* returns a string-representation of a file, e.g.

   ```
   "testexercises.doc, 267 kb, mec"
   ```

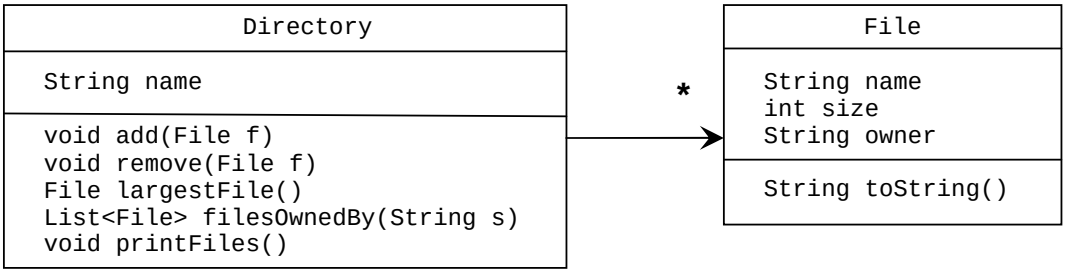


2. Create a *Driver*-class containing an *exam*-method. The method must be static, have return type void and no parameters.

3. Create three *File*-objects, using object references *f*1, *f*2 and *f*3, in the *exam*-method and print out these using the *toString*-method.

Call the observer and demonstrate what you have made so far.

4. Create a new class, *Directory*, representing a directory in a file-system. The class *Directory*, and its relation to the *File* class, is specified in the following UML-diagram:



5. Program the methods *add* and *remove* who respectively adds and removes the *File*-object *f* to/from the Directory-object.

6. Create an object of type *Directory* in the *exam*-method in the *Driver*-class and associate the already created *File*-objects to this object.

7. Program the method *largestFile*. The method returns the larges file in the directory (it can be assumed that the directory is not empty; if two or more files have the same size it is subordinate which file thet is returned). Extend the *File*-class with the necessary get-methods.

8. Use the method *largestFile* in the *exam*-method in the *Driver*-class tio print out information on the largest file in a directory.

Call the observer and demonstrate what you have made so far.

9. Program the method *filesOwnedBy*. The method must return a list of files owned by *s*. Extend the *File*-class with the necessary get-methods.

10. Program the method *printFiles*. The method prints out a list of all files in a directory arranged by size.

Call the observer and demonstrate your final solution.

Implementation of Computer Science in Context - a research perspective regarding teacher-training

Ira Diethelm, Claudia Hildebrandt and Larissa Krekeler
Carl von Ossietzky University
Computer Science Education
26111 Oldenburg, Germany
firstname.lastname@uni-oldenburg.de

ABSTRACT

In order to increase the number of students and professionals in computer science a context-oriented approach for teaching is often suggested, e.g. in [7]. But there is no structured approach to transfer the concept of computer science to its implementation at schools or universities, yet. This aspect is also not mentioned in current research. This paper aims to open a discussion by shifting the focus from the concept to its realization which requires teacher-training and teaching material. We present our initial steps for investigating the transfer of the symbiotic implementation strategy and how it was established for *chemistry in context* to the field of computer science.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer Science Education, Teacher Training

1. INTRODUCTION

Innovative teaching approaches have one problem in common: How to put them into practice. New approaches often sound very promising and helpful for students to learn more CS or motivate them for it. But studies like [6], already made in 1971, showed that top-down strategies are not enough to implement a new approach which aims to change the teaching practice of teachers.

In the case of *CS in context* this would be to motivate the teachers to rearrange their teaching concepts to a much more student centered approach. It points out the relevance of the regarded contents in the context of their every day life. It also aims to enhance teaching methodology. Implementation of this concept appears to be a big amount of work. But we can learn from similar projects of natural sciences such as *chemistry in context – ChiK*. Therefore we start with a report on *ChiK* and its symbiotic implementation strategy in the following section. We also try to explain why we have so few students in our CS studies at universities.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

In section 3 we discuss the special conditions of CS teachers that may influence the success of an implementation of CS in context. There we elaborate several factors which should be considered during a research project for the implementation of *CS in context*. The aspects discussed are also reasons why teacher training in CS is urgently required.

In section 4 we present our first ideas for research questions and research methodology including first items for a survey. We also report on our activities preparing for the survey and some expected results. We hope that our results will help to overcome the gap between theory and practice in teaching CS in context.

2. RELATED WORK

2.1 Computer Science in Context

The phrase *CS in context* is used in differing ways: *CS in context* is used for a courses at university that focuses on more practical aspects of computer science like information and application. Also, a context oriented approach often means to use project oriented teaching methodology.

Context-oriented approaches for schools like *ChiK*, see e.g. [11], are a bit different: They aim at the change of teaching practice for more "authentic science" in class and hence to make it more interesting for students. Therefore, teaching units have to be based on relevant contexts. These contexts should not only be used as a motivation at the beginning but also be present in all following parts of the teaching unit. *ChiK*-units aim to rise the variation of teaching methods and to point out basic concepts (like the relationship between behavior and structure). *ChiK* provides a pattern with 4 phases per unit [11]: 1. phase of contact (e.g. a question or debate): personal relevance, interest; 2. phase of curiosity and planning (e.g. a mind map): identify important questions; 3. phase of elaboration: inquiry, results, presentation; 4. phase of deepening and connecting: reflection, understanding, personal relevance.

We think that this pattern also might fit for CS courses. Koubek et al. [8] suggested an approach of *CS in context* called in German "*Informatik im Kontext - IniK"* that derives from the other context projects. *IniK* also has three aims: 1. orientation on relevant contexts; 2. variety of teaching methods and 3. principles and standards. Some usable teaching units for this approach can already be found in [9].

2.2 Teachers' perspective

Implementing a concept often means to modify the teach-

ers' beliefs and behavior. Therefore, we have to focus on the concept and on the teachers.

Publications in CS research dealing with the teachers' perspective are less common: As one of the later ones Lister et al. analyzed in [10] the understanding of teaching of computing academics and found differing ways that CS academics understand teaching. They suggest "that academics who are aware of the range of understandings will be better able to decide how to design a revision or a new offering." Results of [13] show that "a Conceptual Change/Student-focused approach to teaching is more likely to lead to high quality student learning and to greater teaching satisfaction". And for a context oriented approach one needs to empathize with the students' position, a student-centered belief of teaching. But Pears et al. [12] conclude with: "While much effort has been spent on raising academics' awareness of student-centric approaches to teaching and learning in computer science over the last ten years, it appears that teacher attitudes lag behind. Teachers experience disempowerment in the conduct of their day-to-day teaching practice and there is a risk that this can lead to passivity and disengagement, to the detriment of learning." So, how to bridge this gap?

The first thought to get a teaching approach into practice often is to give a short workshop of a few hours in hope that the attendees will perform in the right way afterwards. Maybe they need another workshop a few weeks later and then the attendees, so the idea, are able to transfer their knowledge to other teachers. But research has shown that an unreflected participation at isolated events is not leading to changes in the teaching behavior in classes. It leads to isolated knowledge only. It is shown already in 1971 by [6] that this relies on a misbelief: "It was assumed [...] that any professional teacher 'worth his salt' could read a document describing the innovation and then, on his own, radically change his behavior in ways that are congruent with the new role model."

One main reason why this top-down strategies fail is that concept level and application level are regarded separately. As a result not enough attention is paid to the special needs of practice like those of the single teacher, see [1].

The fact that teachers' attitudes in CS still lag behind and are difficult to change can be explained as follows: Teachers not only need knowledge of the subject they teach. They need knowledge about the complex activities in class and they need so called *subjective theories* about their work, see [2]. Most publications about teachers' cognitions use some general assumptions that characterize the underlying conception of man. One of them is that teachers not only use knowledge from their formal studies, but from their own time when they went to school and knowledge that comes from their unreflected every-day work at school as a teacher.

According to Dann, [2], p. 166, it takes three steps to change the cognition and subjective theories effectively: The "already existing knowledge and problem-solving capacity has to be activated, ... [then, the] individual subjective theories have to be confronted with new knowledge, [and, finally, to] guarantee that the newly generated knowledge becomes better than the old one, it has to be used."

2.3 Symbiotic implementation

The symbiotic implementation strategy of ChiK, see [11], is characterized by the foundation of learning communities, consisting of 2 teachers from each school, one researcher and one school administrator. Every person in this group of 8-14 persons is regarded as an expert. This communities meet every 6 to 8 weeks. They aim to develop teaching material, try it in their own school and reflect their experiences. So, together they develop a solution that can be shared and fits with the specific conditions and needs of the participating teachers. The personal beliefs and attitudes of the teachers regarding the innovation are crucial.

Most of the groups started from teaching units or materials they already had and began to modify them so that they made an context-oriented unit of it (activation and confrontation according to Dann, above). Then some or all members of the groups tried this material on their own in their classes and evaluated it. In group they reflected on their experiences (to realize that the new teaching method is better than the old one).

Fey et al. evaluated the success of ChiK and summarized their results in [4] after the first year regarding the aims of the project: The perceived **variety** of teaching and learning methods increased. **Co-operation** with teachers of other subjects was enhanced within the participation schools. The perceived relevance of the group meetings correlates significantly with the enhancement of the feasibility of ChiK. The higher the **perceived relevance** of co-operation within these learning communities is, the higher is the perception of a teacher's own **growth of competence**.

In general we can claim that their strategy was quite successful. In the meantime it became a common teaching approach for chemistry in Germany and it is implemented at school, in school books and embedded in the official curriculum. So we think this setting might produce similar results in CS.

3. CONDITIONS OF CS TEACHERS

For a successful transfer and design of an symbiotic implementation strategy for CS in context there are some factors which have to be discussed. Some conditions of CS teachers differ from those of chemistry teachers. But there are no publications known that focus on research of this conditions in CS teacher education or in CS teacher training. So we only can sum up some observations and statistics:

Comparing the number of chemistry or physics teachers we can only find a few CS teachers. There are many schools with only one CS teacher. As a consequence the co-operation and exchange of teaching material, new ideas and feedback among CS teachers is very restricted just because of the distance. This makes **co-operation** a very important aspect for a strategy for implementing CS in context.

Because CS is a quite young discipline the teaching material and curricula for CS at schools are not as uniformized as they are for the natural sciences. The consolidation, tradition in educational research and self-image of CS lags far behind those of natural sciences. Most CS teachers haven't had CS as a subject at school when they were young. This is relevant for the subjective theories of CS of the teachers. So the **perception of teachers for CS or CS teaching** may be an interesting factor as well. We think that these perceptions differ a lot.

There are very different ways to become a CS teacher in Germany: a few studied CS education, some only CS and took no studies in pedagogy, some have been teachers for other subjects, mostly maths, and decided to take additional courses in CS when they were already working at school or

are self-taught. Knobelsdorf and Schulte showed in [7] that some of their students want to become a CS teacher because their own teacher was incompetent and that they had to learn CS on their own. This uncovers another requirement for the design of our implementation strategy: It has to deal with the **differently qualified teachers** and the resulting competences in CS.

An enhancement of teaching competency is shown when teachers try new teaching techniques and methods. And it is shown in the ability to cope with frequently changing conditions, see [5], p. 106. With respect to the fact that CS teachers have to deal very often with changing conditions – a change of tools and of computers they work with at least every 5 years – a perceived low competence becomes plausible.

As a first summary we can say that there are at least 3 important factors for CS teachers that may influence the implementation of new teaching concepts. For the implementation of *CS in context* we have to focus on these factors:

- missing communication and co-operation,
- perception of CS or CS teaching,
- perceived competences and qualification history.

Nevertheless it is another field of research to identify this influencing factors. So we use them as an assumption for our research project.

4. RESEARCH PERSPECTIVE

A transfer of the symbiotic implementation strategy from chemistry to CS seems worth trying and promising.

4.1 Research questions

We assume that the above mentioned factors influence the quality and the success of a symbiotic implementation of *CS in context* and CS teacher training in general. To prove this our research has to answer the following questions regarding teachers:

- What are the initial states of teachers for each factor and how do these values change during a testing period?
- How do teachers perceive the success and quality of the implementation itself?

Additional research questions are similar to those from ChiK [11]:

- Did the teaching behavior change in the designated way?
- What changes are reported by teachers and students as compared with prior teaching and learning experiences regarding context orientation and the variety of teaching methodologies?
- Did the motivation of the students for CS increase?

4.2 Research Methods

To answer these questions we are developing questionnaires for teachers and students. We decided to use online questionnaires that will be used at the beginning of next school year and at the end of each school year for the next 3 years.

To analyze the isolated factors for the teachers from section 3 we will have to find items regarding the following questions:

- How do they describe their exchange with colleagues of CS and other subjects?
- What is their history of qualification for CS teaching?
- How do they perceive their competences?
- What's their teaching practice like?
- How do they deal with the frequent change of conditions of teaching CS?
- Is CS an established subject at their school?
- How many CS teacher colleagues do they have at their school?
- Do teachers perceive an enhancement of their teaching methodologies during the project and do their students perceive this also?

For the research questions regarding the variety of teaching methods and the co-operation in the groups we will use similar tools like ChiK used. Some sample items could be:

In my CS lessons I use a wide variety of teaching methods
My CS lessons is based on questions of topical interest or questions from social or private life.
My CS lessons pass like planned
I am able to identify the problems in my lessons. I am able to cope with them.

The questionnaires will be followed up by an interview study and one study with repertory grids of Kelly to confirm the outcome of the survey more in detail and to be able to get a closer view to the teachers' personality.

For the students' perspective we'll try to evaluate if the interest in CS is increasing and if students want to choose a profession in CS related fields. Sample items for this may be:

I enjoy my CS lessons.
When I find something about CS in newspapers or books I read it through.
I hope to find a job where I have to do with CS every day.

We'll question the students that belong to the teachers of our study and comparable classes that have no CS lessons. We also like to ask students with CS but no *CS in context* in addition to that.

4.3 Sets of teachers

We already set up groups of teachers that take part in our research, so called teacher sets. Now, we will introduce them shortly.

First, we formed a set of 8 teachers of different schools and different school types (from secondary and vocational schools) who want to develop new teaching material and teaching units for the context "energy" in CS classes, see [3]. They meet every 8 weeks and currently they have decided to start with the objective of energy consumption of hardware systems. Their students are 16 years or older.

A second initiative focuses on the technical aspects of CS. Here there are about 20 teachers from 13 schools cooperating to create teaching materials and units for the 7th

grade (13 years old). Most of the teachers want to start planning their units with the context "robotics" and with Lego Mindstorms.

For these two sets the bigger context (energy or technical aspects) was given because they also belong to other research projects regarding these topics. The teachers have just met a few times to organize themselves in the group and to decide which topics to work on first. The third initiative will start in November when secondary school teachers will meet and create teaching units in context of a free choice.

All teachers that we met where asked for their motivation to join our projects and all answered that they want to co-operate and exchange material with other colleagues.

4.4 Expected results

We regard this research project as an initial shot in a set of projects. Due to the fact that we use similar settings and similar research questions like *ChiK* to some points the results may be similar in those questions. But regarding the factors that separate teachers of chemistry from those in CS, we hope to be able to explain differing data with these factors. And we might find a correlation of the co-operation and the perceived competences of the teachers. Besides that we are curious about a connection between the perceived competences and the qualification history.

Altogether, we expect to show that teachers of the sets describe their co-coperation with teachers from other schools as helpful. We think that the co-operation of CS teachers relate to the perceived quality of the work in the set and the implementation of new teaching methods. We also expect to find that students in context-oriented CS classes are a bit more motivated for studying CS or related topics than in the comparing groups of students who had no context-oriented lessons in CS.

5. CONCLUSIONS

The results of research to this questions will effect the design of teacher training units for CS in context and the design of the practical phase of teacher education as well. The results may be also helpful for the design of school curricula and study programs at school or universities. If we use the results maybe we can manage to reduce the obstacles and rise the motivation level to study CS or CS related topics. And teachers may be able to handle the different pre-knowledge level of students much better and use them for the work on a certain context.

On the long run we hope to initiate some more effects: We hope to support teacher training in CS in general and to support the development of basic concepts for CS that can be used to augment CS lessons in the future like it did in chemistry or physics. Teachers who have joined the teacher sets of a symbiotic implementation could create new sets on their own.

We also think that the symbiotic implementation strategy can help to give CS more weight at school just because teachers teach interesting things that help students to understand the phenomena of their every day life.

6. REFERENCES

[1] P. Blumenfeld, B. Fishman, J. Krajcik, and R. Marx. Creating usable innovations in systemic reform: Scaling-up technology embedded project-based science in urban schools. *Educational Psychologist*, 35:149–164, 2000.

[2] H.-D. Dann. Subjective theories and their social foundation in education. In *Social Representations and the Social Bases of Knowledge*. Hogrefe & Huber, 1992.

[3] I. Diethelm and S. Moll. Energy education and computer science. In *WCCE 2009: 9th World Conference on Computers in Education*, 2009.

[4] A. Fey, C. Gräsel, T. Puhl, and I. Parchmann. Implementation einer kontextorientierten Unterrichtskonzeption für den Chemieunterricht. *Unterrichtswissenschaft*, 32:238–256, 2004.

[5] K. Fussangel. *Subjektive Theorien von Lehrkräften zur Kooperation – Eine Analyse der Zusammenarbeit von Lehrerinnen und Lehrern in Lerngemeinschaften*. Dissertation, University of Wuppertal, 2008.

[6] N. Gross, J. B. Giacquinta, and M. Bernstein. *Implementing organizational innovations: A sociological analysis of planned educational change*. Basic Books, New York, 1971.

[7] M. Knobelsdorf and C. Schulte. Computer science in context - pathways to computer science. In *Koli Calling 2007: Proceedings of the 7th Baltic Sea conference on Computing education research*, 2008.

[8] J. Koubek, C. Schulte, P. Schulze, and H. Witten. Informatik im Kontext (IniK) - ein integratives Unterrichtskonzpt für den Informatikunterricht. In *INFOS 2009: Informatik und Schule*, 2009.

[9] J. Koubek and the IniK-Group of Königstein. IniK - Informatik im Kontext, www.informatik-im-kontext.de, 2008.

[10] R. Lister, A. Berglund, I. Box, C. Cope, A. Pears, C. Avram, M. Bower, A. Carbone, B. Davey, M. de Raadt, B. Doyle, S. Fitzgerald, L. Mannila, C. Kutay, M. Peltomäki, J. Sheard, Simon, K. Sutton, D. Traynor, J. Tutty, and A. Venables. Differing ways that computing academics understand teaching. In *ACE '07: Proceedings of the ninth Australasian conference on Computing education*, pages 97–106, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[11] I. Parchmann, C. Gräsel, A. Baer, P. Nentwig, R. Demuth, B. Ralle, and the ChiK Project Group. "Chemie im Kontext": a symbiotic implementation of a context-based teaching and learning approach. *International Jounal of Science Education*, 28(9):1041–1062, 2006.

[12] A. Pears, A. Berglund, A. Eckerdal, P. East, P. Kinnunen, L. Malmi, R. McCartney, J.-E. Moström, L. Murphy, M. B. Ratcliffe, C. Schulte, B. Simon, I. Stamouli, and L. Thomas. What's the problem? teachers' experience of student learning successes and failures. In *Koli Calling 2007: Proceedings of the 7th Baltic Sea conference on Computing education research*, 2008.

[13] K. Trigwell and M. Prosser. Development and use of the approaches to teaching inventory. *Educational Psychology Review*, 16(4):409–424, 2004.

Levels of Awareness of Professional Ethics used as a Sensitizing Method in Project-Based Learning

Tero Vartiainen
Turku School of Economics, Pori Unit
P.O. Box 170
FI-28101 PORI, FINLAND
Tel +358505200794
tero.vartiainen@tse.fi

Ian Stoodley
Queensland University of
Technology, GPO Box 2434
Brisbane, QLD 4001, Australia
Tel 61 7 31382445
i.stoodley@qut.edu.au

ABSTRACT
There is a need for educational frameworks for computer ethics education. This discussion paper presents an approach to developing students' moral sensitivity, an awareness of morally relevant issues, in project-based learning (PjBL). The proposed approach is based on a study of IT professionals' levels of awareness of ethics. These levels are labeled *My world*, *The corporate world*, *A shared world*, *The client's world* and *The wider world*. We give recommendations for how instructors may stimulate students' thinking with the levels and how the levels may be taken into account in managing a project course and in an IS department. Limitations of the recommendations are assessed and issues for discussion are raised.

Keywords
project-based learning, ethics integration, phenomenography, variation theory, awareness

1. INTRODUCTION
Ethics teaching in computing is recognized as a vital part of computing education, for example professional ethics have been incorporated into curricula in the computing disciplines [e.g., 2], frameworks for ethics teaching in computing have been proposed [e.g., 5], text books on ethics education have been published [e.g., 4] and techniques for teaching computer ethics have been proposed [e.g., 1]. In this paper we present a new approach to be used in project based learning (PjBL). We propose that through this approach instructors of a project course would be able to support students' growth in moral sensitivity, that is, their recognition of morally significant issues, and orient the students appropriately towards ethical action. Moral sensitivity is, according to James Rest's [7] Four Component Model (FCM), the first step in developing moral behavior. The FCM describes four simplified and overlapping processes, according to which an individual may fail to act morally. These processes are capabilities which can be focused on in educational interventions. The first process, moral sensitivity, involves awareness of how our actions affect other people. It includes the capability to construct different possible scenarios for moral conflicts and how different actions have an influence over others. After recognizing a moral conflict, one has to solve it, i.e., make a decision concerning what to do. The second process, moral judgment, is about judging which courses of action are the most justified. As moral judgment develops, a person's problem-solving strategies become more directed towards others and more principled in nature. The third process, moral motivation, refers to the importance people place on moral values. Moral motivation is about prioritizing moral action. This speaks of having the will to carry through to action the choices made in the preceding (second) process. A clear example is if someone chooses to lie to maximize profit, although he or she understands that being honest is the moral choice to make, this is a failure in terms of moral motivation. The fourth process, moral character, refers to the psychological strength to carry out a line of action. Courage, perseverance and implementation skills are needed to carry out what a person perceives to be morally right to do. FCM describes four types of failures in moral behavior but also four abilities which develop as an individual matures morally and which can be reinforced by education.

In this paper, we focus on the first aspect of FCM, developing moral sensitivity in students. To do this, we introduce facets of awareness of professional moral behavior into the PjBL environment. We describe five cumulative facets of awareness, called 'citizenships' [10]. We argue that it is possible to support students' development to more comprehensive levels of awareness in the PjBL context, that is to say, to support the development of moral sensitivity in students on the issues relevant to information systems development (ISD). Towards this goal, we discuss the implications of such an approach for various people in project based learning.

2. PROJECT-BASED LEARNING (PjBL)
The project-based learning theory is based on constructivism which espouses the following guiding principles: 1) learning is a search for meaning and meaning is derived from experience; 2) meaning requires understanding wholes and their constituent parts; and 3) meaning that is derived from experience is powerful because it is fundamentally self-referent, it is rooted in personal identity and it views life from the inside in the context of social systems. In constructivism, the situational nature of learning is taken into account and therefore authentic or simulated environments are preferred [e.g., 3]. A study by Tynjälä [12]

Koli Calling '09, October 29 – November 1, 2009, Koli, Finland.

showed that students studying in accordance with constructivism, writing assignments and discussing them in groups, showed more development in thinking skills (classifying, comparing, evaluating and generalizing issues) than students reading books and attending lectures. There are five significant features that distinguish the constructivist approach of project-based learning from other forms of learning [3]:

- a problem or question serves to drive learning objectives;
- constructing a concrete artifact (cf. problem-based learning in which students work on paper cases without a concrete end product);
- learner control of the learning process (pacing, sequencing, actual content);
- contextualization of learning (what we learn in a particular context we recall in similar contexts); and
- projects are complex enough to induce students to generate questions of their own.

PjBL does not inherently require real-world tasks, but at university level such tasks are often utilized to provide students with as authentic an experience as possible. Developing generic skills such as teamwork is an essential element in many models of PjBL. The characteristics of project-based learning and the existence of project courses in IS curricula [e.g., 9] make it a promising possibility to advance students' moral development in terms of FCM [7]. When students construct an artifact, an information system or other IS related development project, it should be natural to consider the production process and the end result from a moral viewpoint. To prompt in-depth reflection, students need to be guided to critically evaluate their own thinking processes.

We now introduce a study which we consider to be a suitable framework for developing students' moral sensitivity in the PjBL environment in computing.

3. COMPUTING PROFESSIONALS' AWARENESS OF ETHICS

An empirical study of 30 IT professionals in Australia revealed that they experienced ethics in terms of their relation to other people [10]. The professionals acknowledged the rights of an ever broadening circle of other people and this influenced how professionals thought about their own rights. The professionals also acknowledged their responsibility for an ever widening circle of other people. Thus, professionals' rights and responsibilities were increasingly defined in terms of others. This expanding awareness of ethics is represented in five 'citizenships': *Citizenship of my world*, *Citizenship of the corporate world*, *Citizenship of a shared world*, *Citizenship of the client's world* and *Citizenship of the wider world*. Table 1 summarizes these citizenships. In the table the beneficiary is what is directly in view when the professional is acting ethically. In other words, it is the intended recipient of the professional's moral act. The act is how the professional expresses their morality. In other words, it is the way the professional works out concretely their ethical convictions. The intention is the outcome the professional desires from their actions. In other words, it is the professional's goal in engaging in the act. The citizenships are described in more detail below.

Table 1: The citizenship categories of IT professionals' experience of ethics [10]

Citizenship category	Beneficiary	Act	Intention
1. My world	Inner circle	Guarding	Self-preservation
2. The corporate world	Corporation	Devolving	Corporation success
3. A shared world	Client and professional	Sharing	Win-win
4. The client's world	Client	Bearing	Client Success
5. The wider world	Humanity	Serving	Do the 'right thing'

Category 1: Citizenship of my world

When experiencing ethics as Citizenship of my world, the professional focuses on themselves and their close circle of friends and associates. They see themselves as defensively guarding their existing rights, with the intention of self-preservation.

in this particular industry there are two things that get you jobs - your security clearance and your reputation. If your reputation is bad you are not going to get jobs… So, I'm not going to sabotage my career for a company that I work for and I've always had that philosophy. (Participant 11)

Category 2: Citizenship of the corporate world

When experiencing ethics as Citizenship of the corporate world, the professional focuses on their employing organization. They see themselves as loyal employees who devolve the responsibility for decisions to their superiors, with the intention of enabling the corporation to succeed.

if you identify risks to the organisation or to a process then you have a duty of care... to your managers to... bring it to their attention... Provided that you have done your job in identifying that risk, addressing possible recommendations. If they choose to ignore those recommendations then you have devolved your duty of care to them (Participant 28)

Category 3: Citizenship of a shared world

When experiencing ethics as Citizenship of a shared world, the professional focuses on themselves and their clients. They see themselves as sharing equally with the client so both of them benefit and neither are unduly disadvantaged, with the intention of achieving a win-win result.

I'd say that's my clearest picture of ethics in IT and again it's more of the win-win. I think we have an obligation to let the customer win and you win. Don't harm yourself but don't harm the customer. (Participant 6)

Category 4: Citizenship of the client's world

When experiencing ethics as Citizenship of the client's world, the professional focuses on their client. They see themselves as bearing responsibility for the client's welfare, with the intention of enabling the client to succeed.

I still think it goes beyond that and it's this ethical obligation to do what is necessary to meet that client's expectations. It's no good building a system that might meet what was specified to the

letter but if it still doesn't work for them or if it's still going to cause them problems, then you've got an obligation to address those. (Participant 2)

Category 5: Citizenship of the wider world

When experiencing ethics as Citizenship of the wider world, the professional focuses on the needs of humanity in general. They see themselves as generously serving others, even those they may not know personally and even to personal disadvantage, with the intention of doing the right thing.

My ethics have caused me at times to pursue certain paths in my career, so they've been an influence on my choices… particularly of who to work for and what to work on, for example I... once responded to a job ad and I found out... that the job was with a company making gaming machines and I decided to decline to even go for an interview because I… didn't feel it'd be ethical… (Participant 9)

These experiences of ethics build on each other. For example, a professional who experiences the client's world does not loose sight of their own world, however the client's world influences how they see their own world. Thus, these are not developmental stages in the sense that the earlier perspectives are left behind as professionals adopt the later perspectives. Rather, they are facets of awareness which are built on and broadened as the professional experiences ethics in an increasingly comprehensive way.

4. INTEGRATING AWARENESS LEVELS INTO PjBL

In this Section we reflect on the application of the insights offered by the Stoodley's study to the PjBL. In PjBL the relation between instructor and students may be very sensitive [13]. To become better aware of such relations, five levels of instructor intervention were defined towards a student group: 1. outsider; 2. observer; 3. inspirer; 4. participant; and 5. decision maker [11]. Ideally, an instructor should stay at observer and inspirer levels to guarantee independent functioning of a project group and to give students the whole responsibility of their own project [13, p. 703]. At the inspirer level an instructor may be able to direct students' attention towards what the students perceive to be ethical aspects of project work and to the wider Citizenship experiences that students could be expected to experience in the client context and engage in dialogue with the students about the implications of those experiences. To avoid indoctrination, imposing a body of doctrines held by the teacher on the student [e.g., 14], the instructor should avoid becoming a participant of the group. This means that the instructor could suggest wider ways of perceiving ethics as represented in the Citizenships, however he or she should not prescribe those wider perspectives. In more concrete terms, the instructor could reflect back to the students the way they seem to be approaching the PjBL situation, then offer an alternative point of view, for example, "*It seems to me that you are looking at this situation from the viewpoint of your group, but what about the client's point of view. Can you think of how they may see this?*" This question offers inspiration to move from *The corporate world* to *A shared world* point of view. For another typical example, in a situation in which an instructor perceives ego-centric behavior among students [13], he or she could say, "*It seems to me that you may not all be committed to the project task and its implementation. If your group belonged to a software house, how would your attitude be tolerated by your supervisor?*" This question stimulates the students to consider moving from the *My world* to *The corporate world* point of view. Given the partial alignment of students' perceptions with professionals' perceptions, it would appear that open discussion of moral issues amongst students in an open forum would bring students into contact with a breadth of viewpoints. Inclusion of the client in such discussion would serve to enhance the possibility of alternative viewpoints to be expressed. It remains for the instructor to offer a supportive environment in which such discussion may take place and to be alert to perspectives which are not being represented, with a view to ensuring these are heard. The Citizenships offer a framework upon which such intervention may be based.

From the viewpoint of managing a project course, there are several ways that the Citizenships can be used. When negotiating with prospective clients the question needs to be asked, "*Does the client maximize the likelihood that students will be exposed to the widest possible range of ethical views?*" Some clients, for example, may only operate within Citizenships 1 to 3, whereas other clients will also embrace Citizenship 4 or even Citizenship 5. Engagement in a project that had benefits to the wider community would be likely to introduce *Wider world* perspectives and if this project was being supported by a corporation then it would also quite possibly introduce *Corporate world* perspectives. Also, to expose students to a full range of Citizenship views may not require the direct involvement of every student with every client, but in a project course community the students could be encouraged to talk with students from the other student groups as well as get to know the clients of the other student groups. Thus, the ideal project chosen as a stimulus for instruction would be one which has the highest likelihood of students confronting their own views of ethics, views which differ from their own and views which represent the widest possible perspectives. For example, a project which would help provide such a stimulus would impact a wide range of people, and require the students to communicate between each other and other stakeholders in order to find solutions.

From the viewpoint of an IS department and the curricula, the department could define a strategy to collaborate with industry in such a way that, as a whole, students were exposed to the full range of Citizenship views over the course of their IS studies. However, the exposure of students to moral argumentation and moral conflict solving skills [8] should not be neglected, in order for all the processes of Rest's FCM to be drawn on. Instructors involved with the project course could be educated to recognize the Citizenship levels in students' deliberation and to react appropriately. In instructor recruiting the capabilities of university teachers for this kind of ethics integration could be assessed.

This approach may be applied immediately in instructor-student interactions. However, it also suggests the possible need for a comprehensive review of the entire educational setting. Our approach may challenge existing educational objectives, since as we understand it what is typically expected in IS curricula is that students adopt the *Shared world* or *Client's world* perspectives. We propose that from an ethical viewpoint, curricula should include *The wider world* perspective

5. EVALUATION

Given the contextualization feature of PjBL (what we learn in a particular context we recall in similar contexts) [3], it is noteworthy that the PjBL environment does not necessarily resemble the business environment and therefore presents a challenging goal for the educational institute which aims to prepare students to confront moral conflicts in the business environment. Given the control exerted by the learner in PjBL [3], it is noteworthy that our proposal aims to take into account the avoidance of indoctrination by giving students the opportunity to make their own decisions (the instructor adopting the role of inspirer). In addition, in PjBL environments the projects should be complex enough to induce students to generate questions of their own [3]. Our proposal is in line with this feature as morality as such is considered complex [e.g., 6]. Therefore, according to our proposal, students are exposed to discussions and thinking which will require them to take into account the complexities of practical morality.

6. DISCUSSION

We propose the following issues for discussion:

1. How may we enable students to appreciate the views of other stakeholders in the PjBL environment?
2. How may we structure the teaching in a computing department to include ethical aspects?
3. How important is it that students gain a 'wider world' perspective during their university studies?
4. How else could the experience categories be used to prompt students to consider alternative viewpoints to their own?

7. REFERENCES

[1] Applin A.G. 2006. A learner-centered approach to teaching ethics in computing. SIGCSE'06, March 1-5, Houston, Texas, USA. Pp. 530-534.

[2] Gorgone, J. T., Davis G. B., Valacich, J. S., Topi, H., Feinstein, D. L. & Longenecker, H. E. Jr. 2002. IS 2002: model curriculum and guidelines for undergraduate degree programs in information systems. Communications of the AIS 11 (Article 1).

[3] Helle, L., P. Tynjälä, and E. Olkinuora. 2006. "Project-Based Learning in Post-Secondary Education: Theory, Practice and Rubber Sling Shots," Higher Education (51)2, pp. 287–314.

[4] Johnson D.G. 2001. Computer ethics. Upper Saddle River (NJ): Prentice Hall.

[5] Martin C.D., Huff, C., Gotterbarn, D., & Miller, K. 1996. Implementing A Tenth Strand in the CS Curriculum. Communications of the ACM, December, Vol 39. No. 12. pp 75-84.

[6] Packer, M.J. 1985. The Structure of Moral Action: A Hermeneutic Study of Moral Conflict. Basel: Karger.

[7] Rest, J. 1984. The Major Components of Morality. In W.M. Kurtines, J.L. Gewirtz (Eds.) Morality, Moral Behavior, and Moral Development. New York: A Wiley-Interscience Publication. 24-38.

[8] Ruggiero V.R. 1997. Thinking Critically About Ethical Issues, Mountain View, CA: Mayfield Publishing Company.

[9] Scott, T.J., Tichenor, L.H., Bisland, R.B.Jr., & Cross J.H. 1994. Team dynamics in student programing projects. SIGSCE 26 (1), 111-115.

[10] Stoodley, I. 2009. IT professionals' experience of ethics and its implications for IT education. *Professional ethics: The IT experience.* Saarbrucken: VDM Verlag Dr Muller.

[11] Tourunen, E. and T. Vartiainen. 2002. "Ethical Issues in Project Learning," Presented: International Conference on Experiential Learning, ICEL 2002 Cultural and Ethical Dilemmas, 1-5 July, Llubjana, Slovakia. (abstract-based acceptance of articles)

[12] Tynjälä, P. 1998. Traditional Studying for Examination vs. Constructivist Learning Tasks: Do Learning Outcomes Differ? Studies in Higher Education 23 (2), 173-189.

[13] Vartiainen T. 2007. Moral Conflicts in Teaching Project Work: A Job Burdened by Role Strains. Communications of the Association for Information Systems Vol. 20 (article 43) pp. 681-711. ISSN 1529-3181.

[14] Warnock M. 1975. "The Neutral Teacher" in M. taylor (ed.) Progress & Problems in Moral Education, Windsor: NFER Publishing Company, pp. 103-112.

Visual Program Simulation Exercises

Juha Sorva
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland
jsorva@cs.hut.fi

ABSTRACT

In this paper, I propose a new kind of assignment for CS1 courses, the *visual program simulation exercise*, which engages beginner programmers to learn about fundamental programming concepts and the dynamics of program execution. I discuss the software support that such exercises require, and list some key points on the design and evaluation of a visual program simulator system. The paper provides as a basis for further discussions on the use of visual simulation for learning about program execution.

Keywords: program visualization, program simulation, simulation exercises, engagement, automatic assessment, CS1

1. INTRODUCTION

In this paper, I propose a new kind of assignment, the *visual program simulation exercise*, meant for teaching beginners about fundamental programming concepts and the dynamics of program execution in an engaging and automatically assessable way.

The paper is structured as follows. In the next section, I establish the pedagogical problem that I am working on. Section 3 describes some related work on visualization systems for CS1. In Section 4, I explore the potential of visual program simulation as an assignment for introductory programming students, Section 5 debates the software support needed to facilitate program simulation exercises. In Section 6, I describe current and future work on building and evaluating a program simulation system. Section 7 briefly concludes the paper.

2. THE CHALLENGE

Some of the difficulties that students have with learning basic programming have to do with their understandings of *the notional machine*, "the general properties of the machine that one is learning to control" [4]. Many students fail to understand the execution model of programs and the relationship between static program code and dynamic execution. In Du Boulay's words, "a running program is a kind of mechanism, and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes" [4].

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

Numerous studies have shown that students have non-viable mental models of many fundamental programming concepts. References and pointers, classes, objects, and constructors are some of the interrelated concepts that have been reported as problematic (see e.g. [16, 17, 18], and references therein).

Constructivism suggests that teachers must take great care to ensure CS1 students have a viable mental model of the computer on a level of abstraction lower than the one primarily used in the course [2]. To really understand *what* a command in code does, a learner also needs to understand something about *how* the computer goes about dealing with the execution of the command. There is a need for teaching methods and tools that help us teach the execution model and program state at a suitable level of abstraction, neither hiding important concepts nor getting bogged down in technical minutiae.

In some large introductory courses, an additional challenge is the need for automated feedback and grading. There is a dearth of systems that would provide automatically assessable exercises for teaching beginners about the notional machine.

Visualization software is one of the techniques that has been used to address this challenge. The next section describes some work on visualization systems relevant to this paper.

3. RELATED WORK

3.1 Visualization Tools and Engagement

Naps et al. [13] proposed an *engagement taxonomy* (ET), which consists of levels of increasing student engagement with a visualization. Myller et al. [12] suggest that the ET applies best to algorithm visualizations in single-learner environments. Myller et al. present an *extended engagement taxonomy* (EET), which applies better to program visualization contexts and collaborative learning. I have summarized the EET below.

NO VIEWING: There is no visualization but only material in textual format.

VIEWING: The visualization is viewed with no interaction.

CONTROLLED VIEWING: The visualization is viewed interactively. Students can control its speed, for instance.

ENTERING INPUT: Students enter input to a program or subprogram before or during execution.

RESPONDING: The visualization is accompanied by questions.

CHANGING: Students directly manipulate or otherwise change the visualization.

MODIFYING: The visualization is modified before viewing, for

example, by changing source code.

CONSTRUCTING: The visualization is created interactively by students from components.

PRESENTING: Students present the visualization to others for feedback and discussion.

REVIEWING: Visualizations are viewed for the purpose of providing comments, suggestions and feedback on either the visualization or the CS content.

A meta-study Hundhausen et al. [6] found that visual representation matters less than what students do with a visualization. Along these lines, Naps et al. hypothesized that the higher up in the ET students engage with a visualization, the better the visualization aids learning [13]. A learner can perform different tasks with the same visualization; it has also been suggested that engaging with a visualization on multiple levels of the taxonomy is beneficial. At least so far, there is only limited support for these hypotheses. (See [19] for a survey of successful experiences with higher levels of engagement.)

3.2 A Couple of Existing Visualization Systems

Jeliot [10] is a *program visualization* system that visualizes the execution of Java programs. It displays control flow, expression evaluation, the call stack, objects and assignment as an animation that the learner can view at a selected speed or manually step by step. Students may interact with Jeliot visualizations on various levels of engagement. They can for instance look at the execution of a given program (CONTROLLED VIEWING in the EET) or they can use Jeliot to visualize programs they wrote themselves (MODIFYING). Results suggest that Jeliot improves programming skills acquisition [3], directs the students' attention to relevant issues [5], and encourages interaction between collaborating students [11].

There are various *algorithm visualization* systems for visualizing algorithms and data structures that operate on a higher abstraction level than the typical program visualization system. The Trakla2 [7] system is characterized by its *visual algorithm simulation exercises*. To complete an exercise in Trakla2, a student starts out with a description of an algorithm and some data. He uses GUI operations to directly manipulate the visual representation of data structures to which the algorithm is applied, with the goal of following the same steps as the the given algorithm. For instance, an exercise presents each student with a pseudocode algorithm for adding values to an AVL tree, and the student uses the GUI to show where and when nodes are added and rotated.

Trakla2 enables students to interact with algorithm visualizations on a relatively high level of engagement[1] while keeping the exercises compact. (No coding required.) Trakla2 has been shown to improve learning (see e.g. [7]) and is popular with students in a number of institutions. Visual algorithm simulation exercises also have various practical benefits. To reduce cheating, the exercises can be tailored to be different (i.e., to have different data values) for each student or group. Students can be allowed to see a model solution on request to help them learn, after which they can no longer submit a solution to the exact same problem, but can simulate and submit using a new, automatically generated data set. Checking student submissions against the model solution enables quick automated feedback and grading.

[1]Trakla2 exercises involve direct manipulation of visualizations, which has been classified as CONSTRUCTING in the ET [13] and alternatively as CHANGING in the EET [12]. For the purposes of this paper, it is only relevant that Trakla2 facilitates engagement levels above VIEWING and CONTROLLED VIEWING.

4. HOW ABOUT PROGRAM SIMULATION EXERCISES?

The Jeliot system has shown that visualization can help learn about program execution and fundamental programming concepts. Potentially even better results could be achieved through assignments where students engage more deeply and in different ways with a program visualization. Mere CONTROLLED VIEWING of a visualization is perhaps not enough. What kind of new visualizations or exercises could help students learn about the dynamics of program execution? What kind of assignments could we give to students that would help them, even require them, to engage with program visualizations on higher levels of the EET, preferably on multiple different levels? Could we give students program visualization-based assignments that could be automatically assessed for quick feedback and automatic grading in large courses and in online education?

Embedding multiple choice questions into animations of program execution is one approach for deepening student engagement, and some work on this has been done in Jeliot [11]. I propose an alternative approach, suggested by the visual simulation exercises in Trakla2.[2] In a *visual program simulation exercise*, a student (or pair of students) would directly manipulate a visualization of the program's execution. The student would be given a small piece of code, and a visualization of computer memory, particularly the stack and the heap and their contents. Given this visualization, and a set of operations they can use, the student would be required not only to figure out how the given program works, but also to demonstrate this knowledge by indicating step-by-step how it happens. For instance, assuming software support, the student could be required to use mouse clicks, drags and drops, or other GUI operations, to indicate where and when some or all of the following simulation steps happen:

- variable creation
- assigning values to variables, including references and/or pointers
- expression evaluation
- object creation in the heap (and other dynamic memory allocation)
- adding frames on the stack
- storing data in a stack frame (including parameter passing)
- returning values from functions or methods
- the flow of control from one instruction to the next
- removal of data from memory (e.g. end of variable lifetime, ignored return values, garbage collection, object deletion)
- class loading

In this type of simulation exercise, the program and a visualization or a set of visual components are given to the students, but 'making the visualization work' is left to the students. The student is encouraged and required to engage with the visualization as they change it by adding variables, assigning values, creating stack

[2]In his 2009 doctoral thesis, Myller also briefly mentions "visual program simulation" as an interesting avenue to pursue in the future when developing Jeliot. [11, p. 50]

frames, and so forth. This places the exercise in the CHANGING or CONSTRUCTING bracket in the EET. (Which bracket depends on interpretation and how much of the visualization the student has to construct and how much is given.) Successfully completing a well-designed program simulation exercise requires the student to understand in a some detail how program execution proceeds and about how the program's data and state are stored in memory. As the program does not run by itself, 'just looking at the graphics' will not work.

Even though the relationship between engagement with a visualization and learning remains unclear, there is clearly one practical benefit from the activities that are higher up in the EET compared to the first few ones. The activities described by the levels NO VIEWING, VIEWING, CONTROLLED VIEWING, and ENTERING INPUT do not translate very well into assessable exercises. Irrespective of what future research reveals about engagement, simulation exercises have the practical benefit of being readily assessable, also by a computer. Given an executable program, a trace of program execution steps can be determined to gain the correct answer to the simulation exercise. With software support, automatic feedback can be given during or after a simulation, and the solution can be graded automatically against the correct answer.

A point about task authenticity needs to be made. A task given to a programming student should be reasonably authentic, that is, it should resemble the activities of programming professionals. Though visual program simulation in the sense outlined above is not something professional programmers do, mental tracing of program certainly is. Visual program simulation exercises can be viewed as a tool-aided, systematic way of tracing programs.

Visual program simulation exercises probably require software support to be practical. This is the topic of the next section.

5. SIMULATOR TOOL DESIGN ISSUES

A good software system for visual program simulation exercises is correct, clear, relaxed, capable of giving feedback, research-based, easy to use, capable of automation, easy to deploy, and highly configurable.

A *correct* program simulator system features no in-built factual errors in the visualization (although, in a simulation exercise, the learner can create erroneous visualizations). Correctness does not imply high precision; abstracting away of detail is quite acceptable.

A simulator must be capable of creating *clear* visualizations that a user familiar with the system can take in quickly, leaving them free to concentrate on the content. This is a significant challenge, particularly in a general-purpose program simulation system. (See Ma et al.'s evaluation of Jeliot compared to a bespoke visualization tool for reference assignment [9].) A program simulator needs to display a number of memory areas and a significant amount of code and data, as well as offering a selection of GUI operations to the user. It is all too easy to create a cluttered visualization that is little more than a tangle of references or pointers.

A good simulation exercise system is *relaxed* in the sense that it allows students to make mistakes rather than preventing incorrect simulation steps. This gives the students a sense of freedom, and prevents the simulation exercise from turning into a meaningless activity where only one (or very few) operations are possible at any given time, and the exercise is solvable through mechanical trial and error. Likewise, the system – even if it does automatic grading – should not punish students immediately for errors made during the learning process. A key aspect of any CS1 course is student motivation (see e.g. [1]), and a relaxed, secure environment can contribute to that as well.

A pedagogically sound program simulator *gives feedback to the student* at an appropriate time and at an appropriate level of detail. For examples of types of feedback in a visual simulation system, see [8].

A *research-based* simulator is designed to draw on results from computer science education research. For instance, research can help by identifying particularly challenging concepts and those aspects of the concepts that have been found to be critical for learning. The system could be designed to allow students to make specific kinds of common mistakes (e.g. allow simulating object assignment by copying each instance variable's value from one object to another) and give feedback that is tailored to correct the misconceptions that these mistakes originate from.

Making a visual program simulator *easy to use* is a major challenge key to the success of a program simulator tool. As discussed above, simulation steps come in many varieties. They need to be mapped to GUI operations in a consistent and intuitive way. The number of different kinds of simulation steps that are needed and the complexity of the visualization are of some significant concern here. Earlier work on visualizing spatial algorithms [15] suggests that it can be feasible to give students simulation exercises that involve fairly complex visualizations and GUI operation semantics.

Simulation exercises becoming too laborsome to complete is a very real concern. To combat this, it must be possible to perform each individual step in a simulation sequence with minimal effort. This means, for instance, that typing names of identifiers during a simulation should be kept to a minimum or eliminated altogether. Ideally, a core set of simple GUI operations are used consistently during a simulation so that a particular GUI action (say, dragging a value) always has the same semantics. Nielsen's popular usability heuristics [14] provide succinct pointers. For example, undoing and redoing must be supported to allow students to experiment with program execution and add to their sense of freedom as they engage with the visualization and inevitably make mistakes.

A program simulator must be *capable of automation* in the sense that once a particular aspect of program execution has been mastered, that aspect can be ignored in later simulation exercises. For instance, once the evaluation of arithmetic and boolean operators has been covered in a simple early exercise, later exercises could be configured so that any occurrences of these operations in the given program are dealt with automatically, and the students need only concern themselves with the learning goals of the later exercise (say, stack frame allocation and return values in a recursive program). If the simulated program is any longer than just a few lines, such elimination of detail is crucial to prevent the simulation task from degenerating into an exercise in tedium.

To aid dissemination to other institutions, a program simulator should be *easy to deploy*. Web technologies as well as integration with other systems (such as Trakla2, Jeliot and IDEs) make the system more convenient to adopt.

A program simulator tool should be *highly configurable* to suit different learning goals, different teachers, and different learners. Examples of settings that teachers and/or students could adjust include the type of feedback given, the programming language used, and the way references are visualized (say, as 2D arrows, as abstract memory addresses, or with a visual metaphor).

6. CURRENT AND FUTURE WORK

At the time of writing, a visual program simulation system is being designed and built, drawing on the guidelines set out in the previous section. A prototype with a large feature set already exists. We will use the prototype to test whether visual program simulation exercises are a practical way for engaging students with a notional machine in an effective and automatically assessable way. In par-

ticular, we would like to find answers to several subquestions:

- Can we make a visual program simulation system easy enough to use to be practical?
- What are students' attitudes towards the exercises and the system?
- How do students perceive what visual program simulation is and what they are doing as they complete a visual program simulation exercise?
- Does visual program simulation help students gain program comprehension skills?
- Which kinds of non-viable understandings of programming concepts can be addressed through visual program simulation exercises?
- How do CS1 teachers react to the concept of visual program simulation exercises and the system prototype?

We are planning to test the system on CS1 students in spring 2010.

7. CONCLUSION

In this paper, I have sketched out a potentially useful kind of exercise for CS1 courses. This paper provides a foundation for discussions of visual program simulation exercises. More specifically, the paper serves as a basis for the discussion questions:

- Is visual program simulation a practical type of exercise for CS1 courses?
- What kind of software support should be used for program simulation exercises?
- How should a visual program simulation system be evaluated?

8. REFERENCES

[1] T. Ahoniemi, E. Lahtinen, and K. Valaskala. Why should we bore students when teaching CS? In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, CRPIT, pages 139–140, 2007.

[2] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[3] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

[4] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[5] G. Ebel and M. Ben-Ari. Affective effects of program visualization. In *ICER '06: Proceedings of the second international workshop on Computing education research*, pages 1–5, New York, NY, USA, 2006. ACM.

[6] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.

[7] A. Korhonen. *Visual Algorithm Simulation*. Doctoral dissertation (tech rep. no. tko-a40/03), Helsinki University of Technology, 2003.

[8] M. Krebs, T. Lauer, T. Ottmann, and S. Trahasch. Student-built algorithm visualizations for assessment: flexible generation, feedback and grading. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 281–285, New York, NY, USA, 2005. ACM Press.

[9] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood. Using cognitive conflict and Jeliot visualisations to improve mental models. In *ITiCSE '09: Proceedings of the 14th annual SIGCSE conference on Innovation and technology in Computer Science Education*, [to be presented].

[10] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 373 – 376, Gallipoli (Lecce), Italy, May 2004. ACM.

[11] N. Myller. *Collaborative Software Visualization for Learning: Theory and Applications*. Doctoral dissertation, University of Joensuu, 2009.

[12] N. Myller, R. Bednarik, E. Sutinen, and M. Ben-Ari. Extending the engagement taxonomy: Software visualization and collaborative learning. *Trans. Comput. Educ.*, 9(1):1–27, 2009.

[13] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

[14] J. Nielsen. Heuristic evaluation. In *Usability Inspection Methods*. John Wiley & Sons, 1994.

[15] J. Nikander, J. Helminen, and A. Korhonen. Experiences on using TRAKLA2 to teach spatial data algorithms. *Electron. Notes Theor. Comput. Sci.*, 224:77–88, 2009.

[16] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3):203 – 221, 2005.

[17] J. Sajaniemi, M. Kuittinen, and T. Tikansalo. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *J. Educ. Resour. Comput.*, 7(4):1–31, 2008.

[18] J. Sorva. Students' understandings of storing objects. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 127–135, Koli, Finland, 2008. Australian Computer Society.

[19] J. Urquiza-Fuentes and J. A. Velázquez-Iturbide. Pedagogical effectiveness of engagement levels – a survey of successful experiences. *Electron. Notes Theor. Comput. Sci.*, 224:169–178, 2009.

Benefits and Arrangements of Bachelor's Thesis

Teemu Tokola
Oulu University Secure Programming Group
Department of Electrical and Information Engineering
P.O. BOX 4500
90014 University of Oulu
ouspg@ee.oulu.fi

Kimmo Halunen
Oulu University Secure Programming Group
Department of Electrical and Information Engineering
P.O. BOX 4500
90014 University of Oulu
ouspg@ee.oulu.fi

Juha Röning
Oulu University Secure Programming Group
Department of Electrical and Information Engineering
P.O. BOX 4500
90014 University of Oulu
ouspg@ee.oulu.fi

ABSTRACT

The Bachelor's degree and the Bachelor's thesis have been recently introduced to the diploma engineer programmes in Finnish universities. This adoption has led to a number of different practices within Finnish technical faculties, with some notable inter-departmental differences. The Bachelor's thesis presents several practical problems, which are related to advisory arrangements, resources, scope, subject and student motivation. To overcome these problems the thesis for Information Engineering students in the University of Oulu is made in groups and all the groups in a given year have the same subject. The introduction of the Bachelor's thesis presents a significant opportunity for improving the Master's thesis process and avoiding associated problems. Thus, the arrangements of the Bachelor's thesis should help students learn the scientific process necessary for the completion of a Master's thesis. The paper discusses the significance and justifications of this arrangement and possible future developments.

1. INTRODUCTION

In the new degree program introduced with the Bologna process [1], the Bachelor's thesis is the culmination of at least three years of work and results in the completion of a first step in the 3-cycle structure (Bachelor-Master-PhD) of higher education. Previously, Finnish universities' technical faculties have offered only the Master's degree before the Licenciate and Doctoral degrees. Even today, the Bachelor's degree is considered a middle-point in completing the Master's-level studies instead of an actual degree.

To answer the demands for speed and quality of highest education, large singular efforts, such as Bachelor's or Master's thesis, should be both demanding and rewarding learning experiences, but should not become an obstacle for timely graduation. Many students feel that the studies prepare them inadequately for the scope and the challenge of the Master's thesis, and that a gap separates it from other studies [12, p.14-17]. Therefore, the Bachelor's thesis presents a significant opportunity for improving study results on Master's -degree level: instead of creating new scientific results, the Bachelor's thesis should train and prepare the student for the Master's thesis and the process to create it.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

2. APPROACHES

The arrangements of the Bachelor's thesis are naturally crucial in ensuring that students derive benefit from it to their Master's theses. Yet, no strict guidelines were given on how the Bachelor's thesis should be organised in the universities. In Finnish universities with technical faculties, a number of practices have been adopted, and they are outlined below. This survey is by no means comprehensive, as considerable differences exist also between departments and between individual laboratories. The information for this study was collected mainly from online materials of the universities' websites and by conducting short telephone interviews with the staff responsible for the organisation of thesis studies. Some information was easily accessible whereas other information required quite extensive searches in the websites and study guides.

The University of Vaasa has adopted a seminar and thesis approach to the thesis writing. The students choose topics from their respective fields and there is an adviser from the department. The work is individual, but it may be a part of a project on which a group of students make their theses. In the study guide it is stated that the thesis should be done within one year after beginning the work. Both practical and purely theoretical works have been accepted.

The Tampere University of Technology (TUT) has a similar approach, but the departments have differing practices on the individuality of the work. Some departments have only individual theses whereas some allow groups to work together for the thesis. There is also a seminar which consists of lectures that contain information on the thesis process. The theses are mainly theoretical literature reviews.

Helsinki University of Technology (HUT) and the Lappeenranta University of Technology (LUT) have decided to give more credit points for the Bachelor's thesis than the others (10 ECTS vs. 8 ECTS). In HUT the thesis consists of individual work together with a seminar. The students select their thesis topics and advisers from a list that the department sets up and the thesis is individual work. In LUT the individuality of the thesis work is emphasised, but groups are allowed. Also the thesis writing time is restricted to one

semester. There is also group advising, where an adviser advises a number of students at the same time and there are two mandatory intermediary milestones for the thesis.

The required length of the thesis also varies: in Vaasa the preferred length is 30-40 pages, in Tampere 20-30, while others do not provide that information on their websites. Similarly, whether a practical part was required was not always clear. In this respect, practices varied also within departments: in HUT under the information engineering department 2 disciplines out of 3 did not require practical work.

The main observation is that despite some common instructions at faculty and department levels, the arrangements and requirements for the Bachelor's thesis are very varied. It is fair to ask, whether such variation is suitable if and when the Bologna process leads to increased mobility between the technical faculties in Finnish universities.

3. BENEFITS AND PROBLEMS

The process of writing a Master's thesis is very important for the timely and successful completion of the thesis work, and several guidebooks [3, 4, 2] for Finnish students and advisers are available. Still, the experience of many students is, that only after completing the thesis, they know how it should have been done. Offering the possibility to practice it would therefore be most beneficial, and this should be the basis of the Bachelor's thesis.

As a second benefit, the Bachelor's thesis could be useful to research and research groups. If the students are presented with an actual research project with both practical and theoretical work, it prepares them for their Master's thesis and increases the time they spend on research before finishing their Master's thesis. This research "try-out" [10, p.206], could help laboratories recruit the most promising students into a research career, as the students will then have a clearer view of what research is about [11]. The undergraduate students can also contribute to the research in a substantial way as noted in [7].

The Bachelor's thesis process presents a number of problems and challenges for the organiser. Ideally, the process should be challenging but still straightforward to complete and it should be beneficial for the future. The authors have identified the following themes, which may be problematic: *Guidance, resources, scope, subject, student motivation* and *arrangements*.

Guidance provided by the adviser of the thesis is crucial in helping the student learn and develop during the writing process and making it an encouraging experience. Bachelor's thesis is the first such thesis assignment, and the amount of guidance should be higher than in the subsequent theses even if the thesis itself is not very extensive [9, p.89]. However, many Finnish universities have a low teacher per student ratio and finding the time and motivation for a task considered less important than advising higher level thesis students, less time-critical than lecturing and other teaching tasks and far less important than own research is a challenge for the adviser. Also, motivation for advising a scientifically very limited work is bound to be limited too - a problem also related to the scope of the work. In one of the interviews, it was stated that finding motivated advisers for Bachelor's theses is very difficult.

As the amount of students is typically high, the total requirement for resources may be very high. Additionally, as the Bachelor's thesis is a new requirement to an established degree program, it often means additional work for the faculty without any additional financial resources. If the Bachelor's thesis includes any practical part, access to laboratory resources and guidance is necessary. Doing the thesis in a research group can be difficult, as the number of research groups is typically much smaller than the amount of undergraduate thesis workers per year. Finding an alternate venue for thesis work can be very hard as the thesis opportunities in the industry or other places outside the academia are not always available.

The scope of the Bachelor's thesis presents problems in a number of different areas. In many universities, the amount of credits given for the Bachelor's thesis is 8 or 10. This translates into about 200-300 hours of work on the thesis. As it is the first major exercise in scientific writing for most students, a lot of time is spent learning the necessary skills and best practices, leaving less time for core thesis work (research, writing and implementation). As no significant scientific contribution should be expected at this level, a single Bachelor's thesis is bound to be very limited, making it less applicable to a research group setting and less appealing for advisers.

If students can freely choose their subject, it means additional strain on the advisory resources, as the subject may not be within the core expertise of the adviser. Due to the large amount of Bachelor's theses and the humble scope of the work, there is a temptation to dismiss rigorous checking of the background literature, thus opening room for plagiarism and failing to provide the student with appropriate advice on selecting and using references. Additionally, the students might choose poor topics, that require too much from them and/or the adviser. Thesis subjects given by the faculty on the other hand present a resource problem, as many different thesis subjects need to be devised.

Student motivation is an important part of any thesis, as they are usually done alone and often without set deadlines contrary to most other studies. Most faculties seem to have proposed hard deadlines, but none admitted strict enforcement of the deadlines in the interview. As a result, students own motivation and ability to work without the social context that has characterised former studies is very important for a successful thesis completion process [12, p.22-26]. Hopefully, encouraging the students to continuous working and learning in the Bachelor's thesis process helps them complete the Master's thesis in a similar fashion.

The Bachelor's thesis process should reflect the Master's thesis process: if it does not, it will not be very useful for the Master's thesis process. For example, if the Master's thesis contains seminars as in some universities, seminars are good learning experiences [12, p.26-34]. Similary, because engineering Master's theses invariably include a practical part, and literature surveys are typically not accepted, the Bachelor's thesis should contain a practical part.

4. COMMON SUBJECT AND GROUP WORK

As noted in the previous section, there are several problems with introducing a Bachelor's thesis. These problems can largely be avoided by two arrangements. These arrangements can only be accepted on the premise that the Bachelor's thesis is an educational tool more than a scientific contribution. The two main ideas are thus *common subject for all students* and *group work*.

A common subject for all students meant a number of

benefits with regards to problems in subject, advising and resources. When selecting only one subject for the students, more effort can be made in selecting a challenging and suitable subject for an undergraduate thesis. In essence, the benefits of a faculty-chosen subject can be achieved without much resource strain. Also sufficient practical resources can be acquired for the student laboratories, as all students can use the same equipment. More benefits can be found in the advising, as a single subject limits the background work that advisers need to do.

Even with these benefits, the large number of Bachelor's theses each year is likely to strip resources from other areas. Additionally, the scope of the work is still bound to be very limited. These problems can be dealt with by introducing group work into the thesis writing process. When working in groups of 3 students, advisers will have more time to guide each individual group. With 8-10 ECTS credits each, 600-900 hours of work for the Bachelor's thesis provides the opportunity to make the thesis subject more challenging and consequently more motivating for the students. As noted in [10, p.206], group work can be significantly more complex than individual projects, and an appropriate project pushes the knowledge level of the students. It is therefore possible to introduce a sufficiently challenging practical part into the work, while increasing the size of the textual work into what equals an average Master's thesis. These merits are quite significant even without mentioning the well-known benefits of group work to learning [6].

Subject selected for a group Bachelor's thesis should have the qualities listed by Hamelink [5] for a cooperative group task: *multiple possible solutions, interesting problem which is non-trivial and challenging, all group members can contribute* and *a variety of skills are required for completion.*

Undergraduate research is considered very valuable for example in the United States [7], and if the topic allows multiple possible solutions, the research part allows the students to choose their solution based on their own skills. Additionally, having multiple possible solutions is very important with a common subject to allow students to exhibit their skills and separate themselves from other groups.

When students are presented with an interesting problem that is challenging but still possible to solve, they are keen to apply the skills they already possess, and motivated to acquire the skills that may be insufficient for solving the problem. Learning through this kind of a setup has been studied under the concept of problem-based learning [8].

Computer science and engineering are ubiquitous in modern society and implementations are needed in practically everywhere. This provides ample opportunities for requiring a variety of skills and interdisciplinarity [6] in Bachelor's thesis projects. This in turn promotes group work in a fashion in which all group members can contribute.

Assessing individuals part in project work is a difficult task [9, p.87-88], and groups could malfunction due to lack of commitment from some group members [10, p.172-173], but as Bachelor's theses are hardly used when assessing individual applicants for academic positions (at least in Finland), it is sufficient to ensure that each individual student has participated in all parts of the work. Groups should be encouraged to tackle problems immediately, lest the teachers have to take action. Making clear that the groups can fire members as in the Roskilde University [6], encourages student groups to perform appropriately.

Mid-course evaluations can help students avoid problems in their chosen way of implementing the program, and the advisers expertise helps them to a path that will result in a reasonable project conclusion [6]. However, evaluations during the course should not affect the final grade - the grade should be based on the final product. This encourages students to seek advice without fear of having their grade affected by initial bad design or poor implementation.

With the Master's thesis, students on one hand are in need of guidance, but on the other hand feel that a too strictly guided thesis becomes more a work of the adviser than that of the student [12, p.34-35] [9, p.28]. As the Bachelor's thesis does not have similar status as the Master's thesis, it provides an opportunity for higher amount of guidance, allowing the students to learn good practices and to see which parts of the work should be developed. This is in line with opinions that there should be more guidance in the lower academic thesis levels [9, p.89]. Additionally, as the students may not be well-aware of the advisory process [12, p.38], the Bachelor's thesis could serve as an introduction to what is the advisers role, allowing students to benefit more from their advisers in the Master's thesis process.

5. CASE STUDY AND EXPERIENCES

The Bachelor's thesis in Information Engineering at the University of Oulu is done during an 8 ECTS credit course Embedded systems project. The work is done in groups of three, using embedded systems that were made available in a laboratory room, all groups having the same topic. In 2008 students implemented a speech synthesiser and in 2009 a speech recognition system. Both assignments had a mandatory network functionality part, and the students had to implement their own user interface. The subject thus demanded a variety of skills, as groups had to program an embedded system, perform the background research and write the thesis.

To make sure that the students worked all the time, there were several deadlines set for the course: 4 deadlines for returning draft versions of the thesis and three deadlines for demonstrating progress in the implementation. The total time for the course was from January to end of May, but extra time of one month was allowed for groups that failed to reach the target functionality in that time. Additionally it was mandatory for the students to keep track of time spent using an online tool. Students generally completed the work within the given timeframe, although insufficient starting skills could mean as much as 300 hours of work for the course. The timetrack also allowed the teachers to monitor the difficulty and progress, and if necessary, react by providing additional help. This option was used in 2009 when one additional software library was provided for the students after it seemed that the groups are falling behind the schedule.

Although no control group was used, it seems that working in groups has had a positive effect on student performance. Out of the 64 groups in the two years, 58 passed the course and most received very good grades on their work. Previously, even though it was substantially smaller (5 ECTS), the course had a clearly smaller passing percentage. This was quite surprising as the thesis requirements and the challenging task was expected to make the course even harder to pass than the previous one.

Even though the course was a lot of work for the advisers,

it seems obvious that with the number of advisers available (3), we were able to provide significantly better guidance with less resources spent than what would have been possible if there had been approximately 90 individual theses with varying topics.

6. DISCUSSION

The experience from these two courses has shown that this is at the moment a good way to conduct the Bachelor's thesis work on our department. However, some areas of further improvement have already been identified. In the future we will experiment with the group advisement method of LUT, enabling the advisers to give more frequent feedback to the groups and guiding the groups to discuss about their challenges with their peers.

A common subject for all groups is an arrangement that has its benefits, but is mainly done because of resource concerns. However, group work with individual topics related to research group interests could present an excellent opportunity for research groups to find new recruits. Experiences in HUT show that finding topics and advisers is one of the most problematic aspects of their Bachelor's thesis process. Perhaps the demand could be matched with the supply better, if the work is done in groups. Also, the motivation of the researchers offering topics could improve, as groups can invest much more time on a given subject, and therefore could provide concrete benefits to researchers.

As more and more students that have taken this new type of Bachelor's thesis also advance to the Master's thesis level we plan to gather information on their success at that level and compare that with the level of achievement of those students that have had their Bachelor's thesis in other departments of our faculty. It would also be interesting to gather information on the views of the students on whether this type of thesis work has helped them on their Master's thesis or not. This would also provide valuable feedback and development ideas for the arrangement of the Bachelor's thesis.

In order to derive more benefits from the current Bachelor's thesis process, its relationship with the Master's thesis process should be made more obvious to the students in our course. Making students more aware of the significance of an appropriate process and of the significance and the possibilities provided by the adviser should give them even better capabilities for succesfully completing their Master's thesis.

7. CONCLUSIONS

In this paper we have described an efficient and effective way of organising Bachelor's thesis work for a large number of students simultaneously with scarce teaching resources. The outcome of the thesis work has been higher than expected. The two main features that set the arrangement apart from other Bachelor's thesis arrangements were compulsory working in groups and the same topic for all groups in a given year. These arrangements have made it possible to arrange a highly motivating course despite the scarcity of teaching resources.

It is our opinion that both the structured guidance of the advisers and peer support have helped students to achieve the goals set forth in the new course. In some of the theses, students express their thanks to both advisers and fellow students alike for helping them through the thesis work. It has also been very encouraging to see both poor and good programmers enjoy their project and have successes in the practical part of the thesis work.

8. REFERENCES

[1] Bologna declaration, 1999.

[2] K. Ekholm. *Tee gradu! Graduntekijän selviytymisopas.* Teknolit Oy, 1997.

[3] J. T. Hakala. *Opinnäyte ja sen ohjaaminen.* Gaudeamus, 1996.

[4] J. T. Hakala. *Graduopas.* Gaudeamus, 2005.

[5] J. Hamelink, M. Groper, and L. Olson. Cooperation not competition [engineering education]. *Proceedings of the Frontiers in Education Conference*, pages 177–179, Oct 1989.

[6] J. V. Mallow. Student group project work: A pioneering experiment in interactive engagement. *Journal of Science Education and Technology*, 10(2):105–113, 2001.

[7] P. Pratap and J. E. Salah. Radio astronomy: A strong link between undergraduate education and research. *Journal of Science Education and Technology*, 10(2):127–136, 2001.

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

[9] E. Viljanen. *Tutkielman tekeminen.* Otava, 1986.

[10] P. C. Wankat and F. S. Oreovicz. *Teaching engineering.* McGraw-Hill, 1993.

[11] K. Ward. The fifty-four day thesis proposal: first experiences with a research course. *J. Comput. Small Coll.*, 20(2):94–109, 2004.

[12] O.-H. Ylijoki and L. Ahrio. *Gradu lähikuvassa.* Tampereen yliopisto, 1995.

Constructive Alignment: How?

Neena Thota
University of Saint Joseph
Nape, Rua de Londres 16, Macau
+853 66814445
neenathota@usj.edu.mo

Richard Whitfield
University of Saint Joseph
Nape, Rua de Londres 16, Macau
+853 66825994
rcw@usj.edu.mo

ABSTRACT
This paper considers how to design an introductory computer programming course using the principle of constructive alignment. We give an overview of the pedagogic foundation of a theoretical model that aligns assessment tasks with cognitive and affective learning outcomes. We then extend the principle of constructive alignment to factor in students' ways of learning to program, to plan learning and teaching activities and to choose learning media. Finally, we adapt the design for an object-oriented introductory programming course.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

Keywords
CS1, object-oriented programming, constructive alignment

1. INTRODUCTION
Constructive alignment [2] involves the explicit formulation of the intended learning outcomes and choosing teaching/learning activities likely to lead to attaining the outcomes. Assessment tasks are also designed to ascertain students' learning outcomes to see how well they match what was intended and for deriving a final grade. Existing research on the constructive alignment of outcomes, assessments and teaching in programming courses is limited to the mapping of outcomes to content and assessment [4]. An explicit cognizance of ways of learning to program, or alignment of programming related educational media with teaching and learning tasks is absent in the literature.

Koli Calling '09, October 29-November 1, 2009, Koli, Finland.

2. DESIGN
We give an overview of the pedagogic foundation of a theoretical model adapted for an introductory objects-first java programming course (Figure 1). Our theoretical model has its roots in constructivism which emphasizes that learners construct knowledge based on their activities. As a learning theory, constructivism has profoundly influenced the teaching of computer science [1]. We draw learning outcomes from a computer science-specific taxonomy [6] that provides a mapping from a set of computer programming activities to the competencies listed in a matrix. The taxonomy differentiates between the ability to understand and interpret code, from the ability to design and build a new product. To maintain the holistic approach to constructive alignment, we also draw on the taxonomy of the affective domain [7] to determine outcomes for programming related affective values. Programming activities are assessed with grade descriptors drawn from the SOLO [3] taxonomy that was devised to match the evolving structural complexity of learning outcomes and learner responses. The taxonomy is therefore applicable to assessment in the cognitive and affective domains.

Phenomenographic research studies that focus on how students learn object-oriented programming [5] have uncovered a complex relationship between the students' learning, the learning environment and the learning approaches. Research also shows that the teaching environment should take into account the variation in approaches and learning [9]. Therefore, we provide for a variety of learning experiences to help students deal specifically with object-oriented concepts.

Computer science education, by necessity, relies on software for teaching and learning. Introductory programming courses now have a gamut of interactive, visualization and collaborative learning media to meet the learning demands of novice programmers. We provide the learning media to give opportunities for discussion, interaction, adaptation and reflection which are crucial elements in the learning process [8].

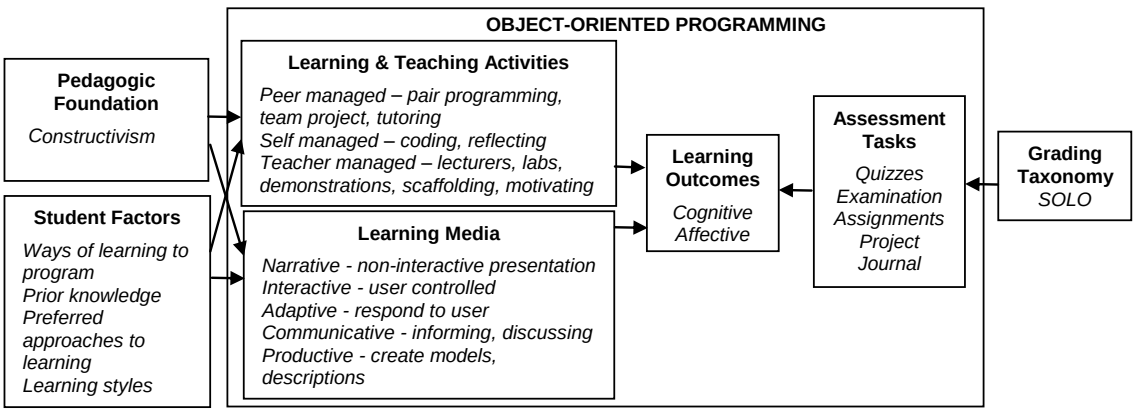


Figure 1. Implementation of constructive alignment.

3. IMPLEMENTATION

Table 1 depicts the specific learning outcomes (cognitive and affective) for an object-oriented programming course and the alignment with the assessment tasks. We then list some teaching and learning activities that are matched with media suitable for students' ways of learning to program, approaches to learning and learning styles.

Table 1. Alignment of outcomes and assessments

Intended Learning Outcome	Programming activities & affective values	Assessment activities
1. Demonstrate knowledge and understanding of essential facts and concepts, relating to object oriented programming	Recognize, trace, implement, translate code	Quizzes & Examination
2. Deploy appropriate theory, practices and tools for problem definition, specification, design, implementation, maintenance and evaluation of programs	Analyze a problem, apply, adapt, relate, present, debug code	Programming assignments & Group project
3. Use object-oriented design as a mechanism for problem solving as well as facilitating modularity and software reuse	Model, design and refactor	Group project
4. Work productively as part of a pair/team	Receiving, responding, valuing	Programming assignments & Group project
5. Demonstrate ability for organization and internalization of values	Organisation, characterization	Journals

In the course design, best practices in object-oriented design, style, documentation and testing are given due importance and misconceptions related to object oriented concepts are explicitly addressed in the learning material and through learning media. Prior knowledge of programming is acknowledged by presenting a range of material suitable for novices and to challenge the more experienced students. Active learning through use of demonstrations and role plays is encouraged. Specific attention is paid to create relevance and increase motivation through the use of constructive feedback. Scaffolding and tutoring are provided for progressive skills development. We interweave the development of affective values in pair programming, team projects and other activities to encourage constructive alignment between the values we want to instill and programming specific attributes.

The design of a programming course using the university specified learning management system augmented by object-oriented content and associated media is reported in a related paper [10]. The use of narrative, interactive, adaptive, communicative and productive media is designed to appeal to different learning styles and to provide a range of learning opportunities. The resources and activities in the learning environment are geared to help students to discern the aspects of variation related to understanding object oriented concepts.

4. CONCLUSIONS AND FUTURE WORK

We have reported on a project to apply the principle of constructive alignment to design an introductory programming course. We have provided an overview of the pedagogic foundation, student learning theories and taxonomies for learning and assessment that underpin the design. We have listed the learning and teaching activities and the use of learning media within the context of an introductory object oriented programming course. We have not reported any formal evaluation of the outcomes. We hope to refine the model and widen the scope to other computer science related courses.

5. REFERENCES

[1] Ben-Ari, M. 1998. Constructivism in computer science education. *SIGCSE Bull.* 30, 1, 257-261. DOI= http://doi.acm.org/10.1145/274790.274308

[2] Biggs, J. 2003. *Teaching for Quality Learning at University: What the Student Does* (3 ed.). SRHE and Open University Press, Philadelphia.

[3] Biggs, J., and Collis, K. F. 1982. *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, New York.

[4] Brabrand, C. 2008. *Constructive alignment for teaching model-based design for concurrency*. In Transactions on Petri Nets and Other Models of Concurrency I, K. Jensen, W. M. Aalst, and J. Billington, Eds. Lecture Notes In Computer Science, Springer-Verlag, Berlin, Heidelberg, 1-18. DOI= http://dx.doi.org/10.1007/978-3-540-89287-8_1

[5] Eckerdal, A. and Thuné, M. 2005. Novice Java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bull.* 37, 3, 89-93. DOI= http://doi.acm.org/10.1145/1151954.1067473

[6] Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., and Thompson, E. 2007. Developing a computer science-specific learning taxonomy. *SIGCSE Bull*. 39, 4, 152-170. DOI= http://doi.acm.org/10.1145/1345375.1345438

[7] Krathwohl, D. R., Bloom, B. S., and Masia, B. B. 1964. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook II: Affective domain*. David McKay Company, New York.

[8] Laurillard, D. 2002. *Rethinking University Teaching: A Framework for the Effective Use of Educational Technology.* (2 ed.). RoutledgeFalmer Press, London.

[9] Suhonen, J., Thompson, E., Davies, J., and Kinshuk. 2007. Applications of variation theory in computing education. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research* (Koli National Park, Finland, Nov. 15 - 18, 2007) 88, 217-220.

[10] Thota, N. and Whitfield, R. 2009. Use of CALMS to enrich learning in introductory programming courses. In *Proceedings of the 17th International Conference on Computers in Education* [CDROM] (Hong Kong, Nov. 30 – Dec. 4, 2009). Asia-Pacific Society for Computers in Education, Hong Kong.

He[d]uristics
- Object-oriented Qualities in Examples for Novices

Marie Nordström
Computing Science
Umeå University
S-901 87 Umeå, Sweden
+4690786 77 08

marie@cs.umu.se

ABSTRACT
The use of examples is known to be important in learning; they should be "exemplary" and function as role models. Teaching and learning problem solving and programming in the object-oriented paradigm is recognised as difficult. Examples should be chosen with care. The object-oriented paradigm is particularly well-suited for the handling of complexity in large systems. This makes the design of pedagogic examples critical, since it is difficult to define concise examples that are still truly object-oriented. There has been no discussion on the quality of examples for novices from an object-oriented point of view, until the recently suggested He[d]uristics. Based on these educational heuristics, a survey of educators' view of object-orientation is planned. A structure for identifying aspects of teaching object-orientation is suggested.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms
Design

Keywords
Heuristics, examples, object-orientation

1. INTRODUCTION
Though largely debated, object-orientation is commonly used for introducing problem solving and programming to novices. How to do this is not straightforward. As educators we have little scientific theory and evidence to support us in deciding on how to introduce object-orientation. The strength of object-orientation lies in the handling of complexity in the design of large-scale system, with high demands on maintenance, efficiency and reusability. The educational situation however, is rather different. Introductory examples are small; the design space is restrained because of the limited frame of reference of the novice, the limited number of syntactical elements available, and the fact that the number of lines of code preferably should be kept to a minimum. To develop examples for novices that will serve as role models for object-orientation, we must rely on established object-oriented principles and practices, such as metrics, heuristics, patterns, code smells and similar concepts proposed by the software community. The difficulty is to design examples that show the strength of object-orientation, and at the same time to avoid overly complex examples that leave the novice behind. If the example fails to, at least, indicate the strength, then novices may conclude that the object-oriented approach introduces complexity rather than contributing to problem-solving, whereas when the example is too complex, students fail to understand the overall big picture. The research-efforts so far, has been to collect empirical data on students misconceptions, common compiling errors etc. An interesting example of discussions, often replacing theoretical approaches, is the discussion on common examples, the '`HelloWorld`'-type, that was initiated in Communications of the ACM [6]. Instead of discussing the object-oriented quality of the example, lots of suggestions were made as how to force this example to be "more object-oriented" [2,3,4]. To address the problem of designing the object-oriented examples for novices we surveyed the literature to establish a set of characteristics for object-orientation in general. Based on these characteristics, a number of heuristics has been suggested for the educational situation to aid educators in designing examples for introducing novices to object-oriented problem solving and programming [5]. The proposed heuristics are called He[d]uristics to emphasize the educational focus.

2. He[d]uristics
The He[d]uristics are targeted towards general design characteristics, which means that more detailed practices, like keeping all attributes private, are not stated explicitly. The particular line of presentation (objects first/late, order of concepts, etc.) or environment used should not affect the object-oriented quality of examples and is not critical to the He[d]uristics. Below the He[d]uristics are presented and described through some of their practical implications.

1. ***Model Reasonable Abstractions***
   - Plausible both from a software perspective and also from a novice perspective.
   - Do not make `main` into the entire program.
   - Real objects (with identity, state and behaviour), e.g. no stateless or behaviourless classes (containers).
   - Small is beautiful (in terms of classes, methods and parameters), e.g. no God classes.
   - Model classes not roles.
2. ***Model Reasonable Behaviour***
   - Separate the model from the modelled.
   - Avoid setter/getters, particularly for attributes.
   - No snippets.
   - No printing for tracing.
3. ***Emphasize Client View***

- Promote thinking in outside expectations.
- Separate the internal representation from the external functionality.

4. ***Favour Composition over Inheritance***
   - How to know which to choose, to avoid the overuse of inheritance.
   - Emphasize the idea of collaborating objects.
   - Delay the introduction of inheritance.
5. ***Use Exemplary Objects Only***
   - Always promote the idea of many objects.
   - No one-of–a-kind classes.
   - Be explicit, do not use static, anonymous classes, and keep the Law of Demeter in mind.
   - Separate `main` from abstractions.
6. ***Make Inheritance Reflect Structural Relationships***
   - Important to distinguish between external and internal is-a relationships.
   - Behaviour must guide the design of hierarchies.
   - Inheritance should separate behaviour.

Some of these characteristics/qualities could be placed under more than one heading. The use of real objects (with identity, state and behaviour) is on one hand a question of abstraction, and on the other the promotion of exemplary objects. In this work ***Use Exemplary Objects Only*** is more intended to focus on the actual presentation of small-scale examples while ***Model Reasonable Abstractions*** is aiming at a proper mindset for object-oriented design. Based on these He[d]uristics an example-evaluating tool has been designed and evaluated [1].

3. EMPIRICAL WORK

The He[d]uristics are formulated to support the object-oriented qualities in an example. To find out how well they match the issues perceived by intended users, educators at different level of teaching, a series of empirical studies are planned.

- Surveys and interviews with educators, both at universities and in upper secondary schools, to investigate their descriptions of what they consider most critical in object-orientation, their view of students' difficulties with the paradigm and important concepts, and how they teach it.
- Testing the He[d]uristics on groups of educators to get feedback on usability.

Making an inventory of teaching practices and examples is necessary to know how to be able to aid educators in their work. How can such a survey be structured? Asking the teacher for his/her personal views of concepts and how he/she addresses problematic issues of learning object-orientation would contribute to the formulation of the He[d]uristics. Figure 1 shows the suggested structure of areas of interest for the first paper/web-administered survey. The results of this survey will be used to fine-tune question-areas and questions for the subsequent survey and interviews. Once the data from these surveys and interviews have been collected and analysed, the He[d]uristics will, if necessary, be adjusted and an empirical study will be submitted to educators to evaluate their usability.

	Teachers personal view on concept	*Teachers view of students difficulties*	*Choice of methodology*
	[C] Characteristical	**[P] Problematic**	**[M] Teaching-practice**
Paradigm (OO)	What are the characteristics of OO? What is most important to stress?	What about OO is most difficult to internalise?	How is OO presented, as paradigm?
Concept (Object)	Ideal objects, how are they defined?	What is perceived as difficult about objects?	How does a displayed object typically look?
Examples	What is typical of a good example?	Does OO-examples differ from examples in other paradigms, to the students?	How are examples chosen and/or designed? What characteristics are prioritised?
Process (OOA&D)	What is characteristic for the problem-solving approach?	What do students find difficult in OOA&D?	How is OOA&D introduced and practised?

Figure 1. Areas of interest structured for survey

4. REFERENCES

[1] Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., and Thomas, L. (2009). *An evaluation of object-oriented example programs in introductory programming textbooks*. To appear in: Rößling, G. and Cunningham, S., editors, ITiCSE-WGR '09: Working group reports on ITiCSE on Innovation and technology in computer science education, New York, NY, USA. ACM.

[2] CACM (2002). Hello, world gets mixed greetings. *Communications of the ACM*, 45(2):11–15

[3] CACM (2005). For programmers, objects are not the only tools. *Communications of the ACM*, 48(4):11–12.

[4] Dodani, M. H. (2003). Hello world! goodbye skills! *Journal of Object Technology*, 2(1):23–28.

[5] Nordström M. (2009). *He[d]uristics—Heuristics for designing object-oriented examples for novices.* Lic. thesis, Umeå University, Umeå, Sweden, Mars 2009.

[6] Westfall, R. (2001). 'hello, world' considered harmful. *Communications of the ACM*, 44(10):129–130.

Recent technical reports from the Department of Information Technology

2010-026 Xin He, Maya Neytcheva, and Stefano Serra Capizzano: *On an Augmented Lagrangian-Based Preconditioning of Oseen Type Problems*

2010-025 Soma Tayamon and Torbjörn Wigren: *Recursive Identification and Scaling of Non-linear Systems using Midpoint Numerical Integration*

2010-024 Elias Rudberg and Emanuel H. Rubensson: *Assessment of Density Matrix Methods for Electronic Structure Calculations*

2010-023 Ken Mattsson: *Summation by Parts Operators for Finite Difference Approximations of Second-Derivatives with Variable Coefficients*

2010-022 Torbjörn Wigren, Linda Brus, and Soma Tayamon: *MATLAB Software for Recursive Identification and Scaling Using a Structured Nonlinear Black-box Model - Revision 6*

2010-021 Michael Thuné and Anna Eckerdal: *Students' Conceptions of Computer Programming*

2010-020 Torbjörn Wigren: *Input-Output Data Sets for Development and Benchmarking in Nonlinear Identification*

2010-019 David Eklöv, David Black-Schaffer, and Erik Hagersten: *StatCC: Design and Evaluation*

2010-018 Jeremy E. Kozdon, Eric M. Dunham, and Jan Nordström: *Interaction of Waves with Frictional Interfaces Using Summation-By-Parts Difference Operators, 2. Extension to Full Elastodynamics*

2010-017 Jeremy E. Kozdon, Eric M. Dunham, and Jan Nordström: *Interaction of Waves with Frictional Interfaces Using Summation-By-Parts Difference Operators, 1. Weak Enforcement of Nonlinear Boundary Conditions*

2010-016 A. Rensfelt and T. Söderström: *Parametric Identification of Complex Modulus*

2010-015 Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine: *Constrained Monotonic Abstraction: a CEGAR for Parameterized Verification*

2010-014 Stefan Hellander and Per Lötstedt: *Flexible Single Molecule Simulation of Reaction-Diffusion Processes*

2010-013 Jonas Boustedt: *Ways to Understand Class Diagrams*

2010-012 Jonas Boustedt: *A Student Perspective on Software Development and Maintenance*

2010-011 Soma Tayamon and Torbjörn Wigren: *Recursive Prediction Error Identification and Scaling of Non-linear Systems with Midpoint Numerical Integration*

2010-010 Maya Neytcheva, Erik Bängtsson, and Elisabeth Linnér: *Finite-Element Based Sparse Approximate Inverses for Block-Factorized Preconditioners*

2010-009 Salman Toor, Bjarte Mohn, David Cameron, and Sverker Holmgren: *Case-Study for Different Models of Resource Brokering in Grid Systems*

2010-008 Margarida Martins da Silva, Teresa Mendonça, and Torbjörn Wigren: *Online Nonlinear Identification of the Effect of Drugs in Anaesthesia Using a Minimal Parameterization and BIS Measurements*

UPPSALA
UNIVERSITET

November 2010
ISSN 1404-3203
http://www.it.uu.se/