Abstraction Ability as an Indicator of Success for Learning Computing Science?

Jens Bennedsen
IT University West
Fuglesangs Allé 20
DK-8210 Aarhus V
Denmark
jbb@it-vest.dk

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

ABSTRACT
Computing scientists generally agree that abstract thinking is a crucial component for practicing computer science.

We report on a three-year longitudinal study to confirm the hypothesis that general abstraction ability has a positive impact on performance in computing science.

Abstraction ability is operationalized as stages of cognitive development for which validated tests exist. Performance in computing science is operationalized as grade in the final assessment of ten courses of a bachelor's degree programme in computing science. The validity of the operationalizations is discussed.

We have investigated the positive impact overall, for two groupings of courses (a content-based grouping and a grouping based on SOLO levels of the courses' intended learning outcome), and for each individual course.

Surprisingly, our study shows that there is hardly any correlation between stage of cognitive development (abstraction ability) and final grades in standard CS courses, neither for the various groupings, nor for the individual courses. Possible explanations are discussed.

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
Abstraction, indicator, success, learning, computing science, CS.

ICER'08, September 6–7, 2008, Sydney, Australia.

1. INTRODUCTION
Abstraction is seen by most computing scientists as a defining characteristic of computing science and abstraction ability is viewed as an indispensable competence for a person to be successful in computing science.

> *Computer Science and Engineering is a field that attracts a different kind of thinker. I believe that one who is a natural computer scientist thinks algorithmically. Such people are especially good at dealing with situations where different rules apply in different cases; they are individuals who can rapidly change levels of abstraction, simultaneously seeing things "in the large" and "in the small".* [D. Knuth]

The above quote by Don Knuth is presented by Juris Hartmanis in his Turing Award Lecture [35]. Hartmanis writes: "One of the defining characteristics of computer science is the immense difference in scale of the phenomena computer science deals with. From the individual bits of program and data in the computers to billions of operations per second on this information by the highly complex machines, their operating systems and the various languages in which the problems are described, the scale changes through many orders of magnitude" (p. 39).

In his Turing Award Lecture *The Humble Programmer* [28], Dijkstra writes: "We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called 'abstraction'; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer." (p. 864).

In *Computing as a Discipline* [23], the authors identify theory, abstraction, and design as the three major paradigms or processes fundamental to the discipline of computing. In the derived Computing Curricula 1991 [29], the three processes are characterized as follows: "Among the three processes —theory, abstraction, and design— it is fair to assume that some undergraduate programs emphasize more theory than design, while others emphasize more design than theory. However, the process of abstraction will normally be prominent in all undergraduate curricula".

Aho and Ullman's imposing textbook *Foundations of Computer Science* [3] was written as an answer to the Denning Report and the derived Computing Curricula 1991 [66]. In the first chapter *Computer Science: The Mechanization of Abstraction*, computer science is characterized as a science of abstraction: "Every other science deals with the universe as it is. The physicist's job, for example, is to understand how the world works, not to invent a

world in which physical laws would be simpler or more pleasant to follow. Computer scientists, on the other hand, must create abstractions of real-world problems that can be represented and manipulated inside a computer." (p. 1).

In [42], Kramer writes: "Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot? Is it possible to improve these skills through education and training? Critical to these questions is the notion of abstraction." (p. 37).

Machanick [46] have described what he calls an "abstraction-first" course idea:

> *The fundamental principle I have attempted to use throughout the design of the new DAA [Data Abstraction & Algorithms] is to expose students to just as much detail as they need to understand a specific concept but no more. The idea is that students cannot be expected to see the point of abstraction if the course itself dumps them into detail indiscriminately (p. 138-139)*

Machanick has also used the abstraction-first approach successfully in an introductory course [45].

Within mathematics education abstraction is also considered important. Richardson and Suinn have developed the Mathematics Anxiety Rating Scale [54]. Ferguson showed that this scale was missing a factor: Abstraction Anxiety [32]. He found Abstraction Anxiety to be high when measured at a community college.

Based on the above it is reasonable to anticipate a strong correlation between abstraction ability and performance in computer science education.

In previous research [9] we specifically studied the correlation between abstraction ability and performance in introductory programming. Most surprisingly we did not find a correlation between the two. In the conclusion we wrote:

> *The result of this study is most surprising. From the outset we were certain that students at a higher stage of cognitive development would get higher scores in the final exam of the introductory programming course. It is not so!*
>
> *There can be several explanations to this. In this programming course coding is prioritized over design. The cognitive requirements are therefore relatively low, and apparently there are other factors that influence the students' success. (p 42)*

Our conjecture that the reason for lack of correlation is due to the characteristics of the introductory programming course suggests that we look for correlation with performance in other courses in the CS curriculum —courses where the cognitive requirements are higher.

Our previous research was restricted to only one course. In this research we report on a three-year longitudinal study to confirm the hypothesis that general abstraction ability has a positive impact on performance in computing science. Our operationalizations of abstraction ability and performance are the same as in our initial study.

We have followed a group of 145 students over a three year period where we have investigated correlation of their performance in ten courses from a CS bachelor's degree with their abstraction ability. This paper reports on our findings from the study. Surprisingly, our findings confirm the result from our previous study [9] since we did not identify a correlation between abstraction ability and performance in learning computer science.

In section 2 we briefly discuss the vast amount of related work on indicators of success; furthermore, we discuss related work on abstraction and abstraction ability in computer science education. In section 3 we present the operationalization of our hypothesis into the two research questions that are addressed in this paper. Section 4 is a description of the experiment design we have applied, and in section 5 we present the results of our statistical analysis. Section 6 is a discussion of our findings and section 7 is the conclusion and a description of potential future work.

2. RELATED WORK

Over the years, lots of potential indicators of success have been investigated; a brief discussion of this strand of research is presented in section 2.1.

Many computer science educators regard abstraction ability as a requirement for competence in computer science; section 2.2 describes in more detail research that address the relationship between abstraction ability and computer science.

2.1 Indicators of Success

In the 1960s, a lot of work on creating and validating psychological tests to select programmers were performed. Much of the work of the Special Interest Group in Computer Personnel Research (SIGCPR) was about psychological tests for the selection of computing staff. At that time not many people were educated in the field but industry had a huge demand for manpower. In 1966 the number of programmers and system analyst was 170.000–200.000; and the number was expected to rise to 400.000 in 1970 [27]. Simpson [60] published a bibliography in 1973 containing 152 publications describing test for programming ability.

A substantial amount of research has been conducted to identify general variables that predict the success of students aiming for a university degree. The variables investigated encompass

- gender [51, 56, 57, 68]
- parents' educational level [65]
- ACT/SAT[1] scores [16, 17, 57]
- performance in prior courses [21]
- emotional factors [10, 52, 63]
- first language [51]
- consistent mental models [20, 25, 26].

Research has been conducted within the general context of education, within computer science and in the more topic specific area of introductory programming [8, 11, 18, 34, 44, 51].

Many of the aforementioned studies present interesting results; however, it is difficult to apply these results to new educational

[1] ACT: is formerly known as the American College Test. An American, nation-wide college entrance exam. It assesses high school students' general educational development and their ability to complete college-level work. It is a multiple-choice test that covers four skill areas: English, mathematics, reading, and science. The Writing Test, which is optional, measures skill in planning and writing a short essay [1]. SAT (formerly known as the Scholastic Aptitude Test and Scholastic Assessment Test) is a standardized reasoning test taken by U.S. high school students applying for college. It covers two areas – verbal and mathematics. [58]

settings where parameters may differ significantly from the context where the findings were observed. Parameters may be different in many respects:

- course material (e.g. textbook, programming language, development environment)
- course structure (e.g. number of lectures and lab hours)
- course work (e.g. mandatory assignments and project work)
- availability of resources (e.g. support material, support for collaboration, and student/instructor ratio)
- the degree of alignment (concordance between syllabus, course content, and assessment)
- type of assessment (e.g. multiple choice, oral, written, and practical)
- instructor (e.g. teaching experience, familiarity with the subject, and personal attitude)
- students (e.g. age, study seniority, and major)
- external factors (e.g. type of institution, nationality, and culture)

This long list of possible variations among courses indicates how difficult it is to generalize findings from one context to another.

The one finding that seems to be most consistent across various investigations —although not strong— is correlation between mathematics score in high school and performance in CS1, but even this result is questionable.

First, we know nothing about the contents and focus of the programming courses where mathematics has been shown to be a predictor of success. Traditionally, programming courses practice programming on problems of a highly mathematical nature (e.g. factorial, radix-conversion, exponentiation, and binomial coefficients). In such cases, it may be the choice of problems rather than programming per se that causes the result.

Second, one might speculate whether the same findings would occur for other high school grades than mathematics; in fact, Rauchas et al. found that "contrary to the generally accepted view that achievement in high school mathematics courses is the best individual predictor of success in undergraduate computer science, success in English at the first-language level in high-school correlates better with actual performance" [53] (p. 398).

Third, Ventura's research [67] did not find math ability to be a significant predictor of success in his introductory objects-first programming course: "the current research calls into question the importance of math in the objects-first CS1. First, there was no correlation between the number of math courses a student took in high school and any of the measures of success in the current study. Secondly, SAT math scores always appeared after measures of effort and comfort level. In the overall models the predictive value of SAT math scores was negligible." [68].

In conclusion, a large number of studies of indicators of success have been conducted and almost all of them have applied a quantitative approach in the positivistic research tradition [24]. Bennedsen [7] and Caspersen [19] present more complete surveys of research in programming aptitude aiming at identifying indicators of success in introductory programming courses.

2.2 Abstraction Ability

Many computer science educators argue that abstraction is a core competency for computer professionals, and that it therefore must be a learning goal of a computer science curriculum — see, e.g., [5, 43, 48, 50, 62].

Nguyen and Wong [48] claim that it is difficult for many students to learn abstract thinking; at the same time they claim abstract thinking to be a crucial component for learning computer science in general and programming in particular. The authors describe an objects-first-with-design-patterns approach to introductory programming with a strong focus on abstract thinking and developing the students' abstractive skills.

In [50], the authors argue that abstraction is a fundamental concept in programming in general, and in object-oriented programming, in particular. The authors describe a four-level ordering of cognitive abstraction activities that students employ when solving a given problem: 1) defining a concrete class, 2) defining an abstract class with attributes only, 3) defining an abstract class including methods, and 4) defining an abstract class including abstract methods. They do not use a general definition of cognitive development, and therefore no standard test instruments that others have proven valid and reliable. Their definitions (and test instrument) are very specific to object-orientation.

Sprague and Schahczenski [62] also argue that abstraction is the key concept for computer science students. They furthermore argue that object-orientation "facilitates, and even forces, a higher level of abstraction than does procedural programming" (p. 211).

Hudak and Anderson [38] found that a measurement of formal operation (the level of formal operations, measured by formal operational reasoning test [55]) and learning style (concrete experiencing, measured via Kolb's learning style inventory [41]) correctly classified 72% of computer science students using a cut-off of 80% or better as a criterion of success in the course. Similarly, Allert [4] and Thomas [64] report that reflective and verbal learners perform better in programming courses than active and visual learners. Contrary to this, a study conducted by Byrne et al. [18] concludes that there is no correlation between programming performance and learning style.

Kurtz [43] used what seems to be a sub-scale of Inhelder and Piaget's stage theory [39] for his study, and found that the measured abstraction level strongly predicted programming success (it predicted 66% of the final score). However, the study's sample was very small (n=23). He found that the first and last developmental levels were "strong predictors of poor and outstanding performance, respectively; and the [developmental] level predicts performance on tests better than performance on programs" [43]. Barker and Unger [6] did a follow-up study (almost a replication but with a shortened version of Kurtz's instrument [43]) with a much larger sample (n=353). They found that the intellectual development level [39] accounted for 11.6% of the students' final grade.

In conclusion: Several studies have focused on the correlation between student performance and abstraction; however, it is difficult to draw a general conclusion from these. Our study will build up knowledge about the impact of abstraction ability on learning computing science.

3. ABSTRACTION ABILITY AND PERFORMANCE IN COMPUTING

Clearly, abstraction and abstract thinking are fundamental concepts in computer science and key components of learning it. For CS education it is therefore mandatory to explicitly aim at the development of the students' abstractive skills. Furthermore, we anticipate general abstractive skills —abstraction ability— to be an indicator of success for learning computer science. Our hypothesis is therefore:

> *General abstraction ability has a positive impact on learning computer science.*

According to the research tradition in this area, we concretize the hypothesis into research questions that enable a quantitative study. We do this by operationalizing the two components of the hypothesis: abstraction ability and learning computer science.

3.1 Quantification of Abstraction Ability

To operationalize the first part of our hypothesis we need to define what we mean by abstraction ability and how it can be measured.

One definition is made by Or-Bach and Lavy [50], who define abstraction ability in terms of object-oriented programming. However, abstraction ability is a much more general skill often defined as part of the cognitive development stage of a person [40].

Our measurement of abstraction ability is based on Adey and Shayer's theory of cognitive development [2, 59]; this theory is a refinement of Inhelder and Piaget's stage theory [40].

Adey and Shayer define eight stages of cognitive development of pupils [2] (Table 1).

Stage	Characterization
1	Pre-operational
2A	Early concrete
2A/2B	Mid concrete
2B	Late concrete
2B*	Concrete generalization
3A	Early formal
3A/3B	Mature formal
3B	Formal generalization

Table 1: *Cognitive development stages*

Adey & Shayer based their stages of cognitive development on a very large research project, CASE, aimed at finding the cognitive development stages of pupils in secondary school [2]. The research showed a different result than the direct connection between age and development stage originally proposed by Piaget, especially in the upper levels. One of the most important results was that only about 30% of the pupils follow the development expected by Piaget.

Based on [40], Adey and Shayer describe what they call "reasoning patterns of formal operations" and group the eight patterns in three groups: Handling of variables, relationships between variables, and formal methods (see [2] for a more exhaustive description). A person can of course be at a higher development stage in one of these reasoning patterns, but "one would not find an individual competently fluent with one or two of the reasoning patterns who would not, with very little experience, become fluent with them all" [2].

Shayer and Adey have used the eight stages for pupils in the age range of five to 16; we intend to use it on students in the age range of 18 to 22. Shayer and Adey found that at the age of 16, 30% of the pupils were at stage 3A and only approximately 10% at stage 3B. Furthermore they found that the curve describing the progression of stages was very flat at that age [2].

We use Adey and Shayer's stage model of cognitive development to characterize the students' abstraction ability.

It is relevant to question whether Adey and Shayer's scale can evaluate the group that is the focus of this study (young people approximately 20 years old). Their study was done in primary and secondary schools in England, whose pupils' age was from five to 16. The concern is that the variation of results for our group of students will be very limited and therefore not useful in the statistical analysis.

Epstein [30] concludes that, in general, only 34% of the eighteen year old persons have reached the formal operational level, so for the general population this scale will give results that can be statistically analysed. Others have used Adey and Shayer's scale or the more coarse-grained scale by Inhelder and Piaget for students in college, e.g., [47, 49]. Nielsen and Thomsen [49] found that many students in Danish colleges have problems with theoretical exercises and found that their cognitive development stage could indeed be measured by Adey and Shayer's test. McKinnon and Renner [47] as well as Hudak and Anderson [38] tested American college students for the Piagetian cognitive development level. We therefore conclude that the scale will be useful for the selected group of first year university students.

In conclusion: We use Adey and Shayer's stage model of cognitive development to characterize the students' abstraction ability. To measure abstraction ability defined in this way, we use a reasoning ability test developed by Piaget and refined by Adey and Shayer for testing at the higher end of the stage model; the so-called pendulum-test [13].

3.2 Quantification of Performance in CS

To operationalize the second part of our hypothesis we need to define what we mean by learning computer science and how it can be measured. In this research we use the results of the final exams of the courses in computer science as a measurement of success.

3.3 Research Questions

Based on the quantifications discussed in the previous subsections, our overall hypothesis is broken down into two research questions: RQ_{overall} and $RQ(g)$:

The first research question address the overall performance in learning computing science measured as the average grade of courses in the bachelor's degree programme:

> RQ_{overall}: Is there a positive correlation between stage of cognitive development and overall performance in learning computing science?

To get a more detailed picture of the lay of the land, we also want to investigate the performance in specific groupings of related courses. Thus, our second research question is parameterized with a variable describing a grouping of related courses:

RQ (*g*): Is there a positive correlation between stage of cognitive development and performance in courses in the grouping *g*?

There are three specific groupings we want to investigate: Content-based grouping, SOLO[2] level grouping, and atomic grouping. The notion of grouping is discussed further in section 4.4.

4. EXPERIMENT DESIGN

This chapter describes the design of the experiment we have conducted. It describes the test instrument used for measuring abstraction ability, the students and courses involved, groupings of courses, the marking scale used, and the statistical analysis.

4.1 The Test

Shayer and Adey have developed several tests to determine the students' cognitive stages. These tests focus on several of the reasoning patterns, but because "the students with very little experience, become fluent with them all" we find it sufficient to use only one test. We use the so called "pendulum test"; a test that has been used for a long time to test young persons' understanding of the laws of the physical world [2]. Shayer and Adey argues that the pendulum test is particular focused on testing the cognitive development stages from 2B to 3B [2], the span of cognitive stages we find relevant for our target group.

The pendulum test consists of questions focusing on the relation between the weight of the pendulum, the length of the wire and the force used to make the pendulum swing. From two concrete experiments (e.g. short wire, small push and heavy pendulum) the students shall try to understand the relation between these three variables and suggest further experiments to make a firm conclusion.

The students volunteered to participate in the test. It was given to them in a lecture hall in the first week of their introductory programming course. They were all informed that the outcome of the test would not be exposed to the lecturers before the exam.

Along with the test there is a detailed rubric for scoring the test. The actual evaluation of the test was done by a research assistant. Based on this —and the fact that it was a multiple choice test— we have confidence in the evaluation of the cognitive development level.

4.2 The Students

The students in this study were participants of the introductory programming course at the University of Aarhus during the academic year 2005-2006.

4.3 The Courses

The courses used in this study are the mandatory courses in the bachelor's degree in computer science at the University of Aarhus, Denmark.

The general structure of an academic year at the Faculty of Science at the University of Aarhus is four quarters (each of seven weeks), each followed by a two- to four-week examination period. Students take three courses (each of five ECTS[3]) in each quarter. The students need to pass individual exams of all courses in order to pass the bachelor's degree programme in computer science. Some of the courses require that the students have passed an exam in a previous course (e.g. *Algorithms and Data Structures 2* requires that the student have passed *Algorithms and Data Structures 1* which in turn requires that the student have passed *Introduction to Programming*). Therefore, there will typically be fewer students participating in the later courses than in the first courses. The drop out rate in the first year is approximately 20 %. Table 2 presents the bachelor's degree program in computer science (the white boxes represent elective courses).

First year			
1.	Introduction to Programming	Perspectives on CS	Calculus 1
2.	*Programming 2*	*Usability*	*Calculus 2*
3.	Algorithms and Data Structures 1	*Web Technology*	*Computer Architecture*
4.	*Algorithms and Data Structures 2*	*Programming Languages*	*Regularity and Automata*
Second year			
1.		*Computability and Logic*	Databases
2.	*Software Architecture*		
3.			
4.			
Third year			
1.	No mandatory courses. The students select among a number of elective courses.		
2.			
3.			
4.			

Table 2: *Bachelor program in computer science*

The courses in italics are those with a grading scheme specific enough to be analysed. Calculus 2 is a math course that is required for the computer science majors to take.

Table 3 gives a short description of the content of each course included in this study[4]. The code in parentheses after the course name indicates the corresponding course in CC2001 [29] in those cases where such a course exists.

Course	Content
Programming 2 (CS111O)	Language concepts (polymorphism, events, exceptions, streams, and threads); Program design; (Recursive) Data structures; Class hierarchies; Frameworks
Usability (CS350)	Human machine interaction; UI components; Interaction; UI tools; Usability methods
Calculus 2	Directional derivative, gradient vector, differentials of functions in several variables; Double integrals over general domains and in polar coordinates; Taylor polynomials

[2] SOLO: Structure of Observable Learning Outcome, see [12]

[3] ECTS [31]: European Credit Transfer and Accumulation System. A full year of study is 60 ECTS points.

[4] Course descriptions can be found at http://mit.au.dk/coursecatalogue/index.cfm?elemid=9045.

	and -series; Eigenvalues and diagonalization; Vector spaces and inner product.
Web Technology	Central technologies related to semi-structured data, communication protocols, and programming of web applications: HTML and stylesheets, XML and schema languages, transformation of XML data, and web services
Computer Architecture (CS220)	Hierarchical computer architecture (digital level, micro architecture level, conventional level, and OS level); Assembly language; Hardware architecture; External devices
Algorithms and Data Structures 2 (CS210T)	Algorithmic paradigms (divide and conquer, dynamic programming, greedy algorithms); Graph algorithms (graph traversal, topological sorting, spanning trees, shortest paths, transitive closure); Text processing (pattern matching, tries, text compression, text similarity)
Programming Languages (CS344 and CS345)	Functional programming (higher order functions, lazy evaluation, polymorphism, modules); Logic programming (unification, back-tracking, knowledge representation, logic grammars)
Regularity and Automata	Formal models of regularity (finite automata, regular expressions, regular grammars); Proof techniques (invariance, structural induction); Applications in CS
Computability and Logic	Models for computation; Computable and semi-computable problem classes; Unsolvable problems; Propositional logic, predicate logic, program logic, and logical proof systems. Gödel's completeness and incompleteness theorems.
Software Architecture	Software architecture and quality attributes; Responsibility-driven design; Design patterns, frameworks, and variability management; Techniques and tools for testing; Tools for large system development.

Table 3: *Description of the courses*

4.4 Groupings

As mentioned briefly in section 3.3, there are three specific groupings we want to investigate: The first is a content-based grouping of courses; we identify a partition of courses based on formalism, technology, and design and implementation. The second is a grouping based on SOLO level of courses; each course is assigned a SOLO level derived from the learning outcomes for the course. The third is the atomic grouping where we investigate the correlation between stage of cognitive development and performance in the individual courses in the bachelor's degree programme.

In the content-based grouping, courses are grouped into one of three groups based on the content of the courses. The formalism group consists of courses with a strong emphasis on mathematical formalism and theory of computing: *Calculus 2*, *Regularity and Automata*, *Computability and Logic*, *Algorithms and Data Structures 2*, and *Programming Languages*. The technology group consists of courses that specifically emphasise technology: *Web Technology* and *Computer Architecture*. The design and implementation group consists of the two courses *Programming 2* and *Software Architecture*. The remaining course, *Usability*, is eliminated from this grouping because its nature is very different from the rest of the courses.

The SOLO taxonomy (short for Structure of Observable Learning Outcome) originates from the study of student learning outcomes in university teaching by Biggs and Collis [12] carried out in the early 1980s. The taxonomy distinguishes five different levels of learning outcome according to the cognitive processes required by students in order to obtain them: (1) the pre-structural level, (2) the uni-structural level, (3) the multi-structural level, (4) the relational level, and (5) the extended abstract level.

Brabrand and Dahl [14, 15] have analysed the courses of the bachelor's degree programme in computing science by careful assignment of a SOLO level to each of the verbs in the description of the intended learning outcomes for the course. The resulting SOLO average for each course is described in table 4.

Course	**Average SOLO level**
Regularity and Automata	3.14
Usability	3.17
Calculus 2	3.20
Computability and Logic	3.25
Web Technology	3.25
Algorithms and Data Structures 2	3.29
Computer Architecture	3.50
Programming 2	3.70
Software Architecture	3.79
Programming Languages	4.00

Table 4: *SOLO level of the courses*

Based on SOLO levels we have split the courses into two groups: Courses with a SOLO average below 3.50 and those with a SOLO average above 3.50. Somewhat arbitrary, *Computer Architecture* is placed in the upper group.

Finally, in the atomic grouping we investigate each course in isolation.

4.5 Data

Information about the score of final exam comes from the administrative system of the university. The data was collected in February 2008.

In general, the grading a student receives is solely determined by the exam at the end of the course, i.e. no marks for assignments, quizzes or other activities during the course are part of the final exam score. In some courses satisfactory performance in mandatory assignments during the course is a prerequisite for attending the exam.

In Denmark, two scales of marks are used: A binary pass-fail scale and a more fine-grained scale. In 2007, the official scale in Denmark was changed from a scale with ten marks (00, 03, 5, 6, 7, 8, 9, 10, 11, and 13) to an A-F scale[5]. Some exams have used the old scale and some have used the new. Consequently, we have used the official conversion from the old scale to the new scale. Table 5 shows the conversion.

[5] with two different fail marks: Fx and F

"Old" Danish scale of marks	ECTS scale of marks	North American scale of marks
11 and 13	A	A
10	B	B
8 and 9	C	C
7	D	C
6	E	D
03 and 5	Fx	F
00	F	F

Table 5: *Conversion between scales of mark*

A student can fail an exam and then later take a re-exam and pass the course. For all courses we have used the most recent score; we have not taken the number of failed exam attempts into account.

Students have three attempts to pass any given exam. They are allowed to withdraw from an exam. Students who have withdrawn are not part of the population for a given course. Students who did not show up for an exam (but did not withdraw) are given "F" as their score.

4.6 Statistical Analysis

In order to check for a correlation between the score of an exam and the cognitive development stage, we have used a Pearson correlation coefficient test. In order to claim a significant correlation, we require that the confidence level ("p") is less than 5% (i.e. there is less than five percent chance of the null-hypothesis —no correlation). A Pearson correlation coefficient is a number between -1 and 1. 1 or -1 indicates a strict linear relation between the variables; 0 indicates no correlation at all. Provided that p is below 5%, a correlation is present if the Person correlation coefficient is numerically above 0.3.

Before performing this type of empirical quantitative research, it is important to consider the required sample size so that the risk of rejecting a hypothesis when it actually is true (called a type II error, denoted ß), or the other way around (called a type I error, denoted α) is acceptable. In order to estimate the required sample size, we consider three concepts [22]:

- **effect size** (f^2), or the salience of the treatment relative to the noise in measurement
- **alpha level** (α, or confidence level), or the odds that the observed result is due to chance
- **power** (1-ß), or the odds that you will observe a treatment effect when it occurs

Having α=0.05, 1-ß=0.8 and $f^2 = 0.15$, we can calculate the required sample size to be 54 [61]. Given that there are over 100 computer science students, it seems more than reasonable to perform the statistical analysis.

145 participated in the pendulum test. They are representative of the overall student group with respect to mathematical skills, age and gender. Some of the students participating follow other study programs, so the number of students who have taken the pendulum test and the exam in a given course may be considerably smaller than 145. 121 students have taken at least one of the exams mentioned in table 3.The exact numbers are described in table 7.

The distribution of the observed cognitive development level as measured by the pendulum test is a number between 5 and 10. The actual distribution can be seen in Figure 1.

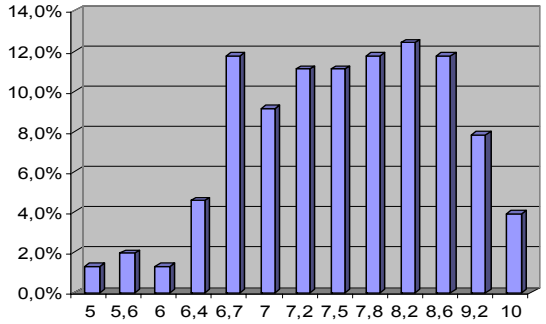

Figure 1: *The distribution of the cognitive development level*

A Kolmogorov-Smirnov test verifies that the distribution can be described as normal. The test instrument used did give scores for cognitive development useful for statistical analysis.

5. RESULTS OF STATISTICAL ANALYSIS

This section describes the results of the statistical analysis.

5.1 Abstraction and Learning Computer Science

The first research question, $RQ_{overall}$, focuses on the overall relation between students' abstraction ability and their performance in learning computer science.

Based on a Person correlation test between the average exam score and the measured abstraction ability, we must conclude, that we can not find a correlation between those (p=0.187). 64 students have participated in the pendulum test and taken at least one core computer science course (i.e. students who only have taken *Calculus 2* are excluded). If we restrict the population to those who have participated in all exams we find the same picture (p=0.26, 25 students have participated in all exams)

The answer to $RQ_{overall}$ is no, thus we must reject the hypothesis of a positive impact of abstraction ability on the overall performance.

5.2 Groupings

The second research question, $RQ(g)$, focuses on the relation between students' abstraction ability and performance in different groupings of courses.

We have investigated the positive impact for two groupings of courses, a content-based grouping (*Formalism*, *Technology*, and *Design and implementation*) and a grouping based on SOLO levels of the courses' intended learning outcome (*High SOLO* and *Low SOLO*).

For all groupings, except a few of the individual courses, the confidence level was far above 5% meaning that the type I error was unacceptable. Consequently, we must reject the hypothesis of correlation between abstraction ability and performance in various groupings of courses.

Only for two individual courses, *Algorithms and Data Structures 2* and *Regularity and Automata*, did we find an acceptable confidence level. For *Algorithms and Data Structures 2*, there is a cor-

relation; for *Regularity and Automata* there is an indication of a correlation.

The answer to *RQ*(*g*) is no, and thus we must reject the hypothesis of a positive impact of abstraction ability on the overall performance.

Table 6 and 7 summarize the results of the statistical analysis; **n** indicates the number of students who have taken the exam, **p** the confidence level and **R** the Pearson correlation quotient.

Grouping	n	p	R	Correlation
Formalism	35	0.136	N/A	–
Technology	33	0.640	N/A	–
Design and implementation	35	0.113	N/A	–
Low SOLO	28	0.456	N/A	–
High SOLO	32	0.149	N/A	–

Table 6: *Correlation between abstraction ability and grouping*

Course	n	p	R	Correlation
Programming 2	66	0.21	N/A	–
Usability	56	0.41	N/A	–
Calculus 2	50	0.28	N/A	–
Web Technology	34	0.45	N/A	–
Computer Architecture	57	0.67	N/A	–
Algorithms and Data Structures 2	45	0.003	0.423	+
Programming Languages	38	0.67	N/A	–
Regularity and Automata	43	0.06	0.290	(+)
Computability and Logic	40	0.43	N/A	–
Software Architecture	35	0.14	N/A	–

Table 7: *Correlation between abstraction ability and individual courses*

6. DISCUSSION

In this section, we will discuss the surprising result of our analysis. The discussion is structured around three themes: (1) Relationship between the SOLO level of the learning goals and abstraction; (2) the teaching of, use of and requirements for abstraction in the curriculum; (3) our quantification of abstraction.

6.1 SOLO vs. Abstraction

It seems reasonable, that students who do well in courses where the expected learning outcome is at the relational level (or the multistructural level) show better abstraction ability than other students. However, we could not verify that performance in courses with a high average SOLO level learning outcome correlates with abstraction ability.

Of course, SOLO levels are not absolute. It might be the case that some courses have as a prerequisite that the student is capable of reasoning at a high level of abstraction. In this case, a learning outcome where the student must be able to give an overview of some concepts (SOLO level 3) based on this abstract reasoning might require more abstraction ability than a course where the student must be able to evaluate (SOLO level 5) some concrete phenomena, say computer programs. However, it is still surprising that we do not find any correlation at all between abstraction ability and performance in courses with a high average SOLO level.

6.2 Abstraction as Product, not as Process

A closer inspection of learning outcomes of courses and assignments in the final examination reveals a pronounced degree of *use* and *modification* of existing abstractions rather than *creation* of new abstractions. To the extent that abstraction is a learning goal at all, it is abstraction as product, not as process. Abstraction ability as such, the ability to abstract and create new abstractions, is at best an implicit learning goal derived from repetitive use and modification of abstractions provided by the teacher.

One example is the course on Computer Architecture; the course objectives of this course are listed in Table 8.

Course objectives for Computer Architecture	SOLO level
explain the organization of computers as multiple levels of virtual machines	4
describe the individual levels	3
apply instruction sets as interface between hardware and software	3
explain the structure of large and small networks of computers	4

Table 8: *Objectives for Computer Architecture*

These objectives do not seem to be related to the creation of abstraction but rather to the use of already defined abstractions (e.g., the levels of virtual machines). One could speculate that the bachelor's programme mostly focus on the use of already given abstraction and not on the creation of new abstractions.

6.3 On Quantification of Abstraction

A third potential explanation for the lack of correlation between abstraction ability and performance is the concrete instrument used to assess abstraction ability.

The validity of the instrument is debatable at two levels. It is debatable whether abstraction ability can be measured as stage of cognitive development and it is also debatable whether cognitive development can be measured by the specific test instrument we have applied. As discussed in section 3.1 and 4.1, the latter is generally accepted as valid.

It is harder to justify the validity of the former: whether abstraction ability can be measured as stage of cognitive development —in this case, the ability to control independent variables in a reasoning task. Abstraction ability is a wide concept which covers much more than the ability to perform sound reasoning. It could be that reasoning ability and variable control is not a prominent competence in the curriculum. However, this does not seem to be the case —on the contrary. The CS curriculum at University of Aarhus is rather formal and theoretical and a core competence in several courses is that of conducting rigorous mathematical proofs about formally defined models and abstractions: Algorithms, automatons, grammars, and models of semantics. This is specifically the case for the courses *Calculus 2*, *Regularity and Automata*, *Computability and Logic*, *Algorithms and Data Structures 2*, and *Programming Languages*. And it is indeed among these

courses that our research hints at some correlation: There is a correlation for *Algorithms and Data Structures 2* and an indication of a correlation for *Regularity and Automata*.

7. CONCLUSIONS AND FUTURE WORK

We have reported on a three-year longitudinal study to confirm the hypothesis that general abstraction ability has a positive impact on performance in computing science.

We have investigated the positive impact overall, for two groupings of courses (a content-based grouping and a grouping based on SOLO levels of the courses' intended learning outcome), and for each individual course.

Surprisingly, we did not find a correlation between abstraction ability and overall performance in learning computing science. Neither a closer inspection of a content-based grouping and a grouping based on SOLO level of courses could demonstrate a correlation between abstraction ability and performance in courses within the groupings. Investigation of correlation for ten individual courses revealed correlation in one case only (*Algorithms and Data Structures 2*) and a weak correlation in another case (*Regularity and Automata*).

Interviews with educators and inspections of textbooks' definition of abstraction indicate that there are only vague and/or very general descriptions of what abstraction is and how it applies to computer science. With the exception of specific applications of the term in, say, abstract data types and abstraction mechanisms in programming languages, there is a general lack of explicit characterization and addressing of abstraction not to mention development of abstraction as a competence, the teaching and learning of abstraction ability. In this light it seems appropriate to embark upon a closer and qualitative inspection of the notion of abstraction and its role in computer science and computer science education. Work in this area is ongoing within mathematics education, see e.g. [33]; there are a few examples within computer science [36] and software engineering [37].

8. ACKNOWLEDGEMENT

It is a pleasure to acknowledge Jens Holbech for his help and assistance in measuring reasoning ability (abstraction ability). Also, we would like to thank all the students who volunteered for this study.

9. REFERENCES

[1] ACT. ACT Assessment, August, 31, 2006.

[2] Adey P. and Shayer M. *Really raising standards. cognitive intervention and academic achievement.* Routledge, London, United Kingdom, 1994.

[3] Aho A. V. and Ullman J. D. *Foundations of computer science.* Computer Science Press, New York, New York, United States, 1992.

[4] Allert J. Learning style and factors contributing to success in an introductory computer science course. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on,* 2004, 385-389.

[5] Alphonce C. and Ventura P. Object orientation in CS1-CS2 by design. In Anonymous *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education.* Aarhus, Denmark. ACM Press, 2002, 70-74.

[6] Barker R. J. and Unger E. A. A predictor for success in an introductory programming class based upon abstract reasoning development. In Anonymous *SIGCSE '83: Proceedings of the fourteenth SIGCSE technical symposium on Computer science education.* Orlando, Florida, United States. ACM Press, 1983, 154-158.

[7] Bennedsen J. Teaching and Learning Introductory Programming – A Model-Based Approach. PhD Thesis. University of Oslo, Norway, 2008.

[8] Bennedsen J. Teaching Java programming to media students with a liberal arts background. *Proceedings for the 7th Java & the Internet in the Computing Curriculum Conference (JICC 7),* London, United Kingdom, 2003.

[9] Bennedsen J. and Caspersen M. E. Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bull*, 38, 2, 2006, 39-43.

[10] Bennedsen J. and Caspersen M. E. Optimists Have More Fun, But Do They Learn Better? – On the Influence of Emotional and Social Factors on Learning Introductory Computer Science. *Computer Science Education*, 18, 1, 2008, 1-16.

[11] Bergin S. and Reilly R. Programming: factors that influence success. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education.* St. Louis, Missouri, United States. ACM Press, 2005, 411-415.

[12] Biggs J. B. *Teaching for quality learning at university. what the student does.* Open University Press, Maidenhead, United Kingdom, 2003.

[13] Bond T. G. Piaget and the Pendulum. *Science and Education*, 13, 4-5, 2004, 389-399.

[14] Brabrand C. and Dahl B. Constructive Alignment and the SOLO Taxonomy: A Comparative Study of University Competences in Computer Science vs. Mathematics. *Proceedings of the 7th Conference on Computing Education Research,* Koli National Park, Finland, November 15-18, 2007, 3-20.

[15] Brabrand C. and Dahl B. Using the SOLO Taxonomy to Analyze Competence Progression of University Science Curricula. *Submitted for publication,* 2008.

[16] Brooks J. H. and DuBois D. L. Individual and environmental predictors of adjustment during the first year of college. *Journal of College Students Development*, 36, 1995, 347-360.

[17] Butcher D. F. and Muth W. A. Predicting performance in an introductory computer science course. *Communications of the ACM*, 28, 3, 1985, 263-268.

[18] Byrne P. and Lyons G. The effect of student attributes on success in programming. *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education.* Canterbury, United Kingdom. ACM Press, 2001, 49-52.

[19] Caspersen M. E. Educating Novices in the Skills of Programming. DAIMI PhD Dissertation PD-07-4. University of Aarhus, Denmark, 2007.

[20] Caspersen M. E., Bennedsen J. and Larsen K. D. Mental Models and Programming Aptitude. *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education.* Dundee, Scotland. ACM Press, 2007, 206-210.

[21] Chamillard A. T. Using student performance predictions in a computer science curriculum. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.* Bologna, Italy. ACM Press, 2006, 260-264.

[22] Cohen J. *Applied multiple regression/correlation analysis for the behavioral sciences.* Lawrence Erlbaum Associates, Inc, Mahwah, N.J., 2003.

[23] Comer D. E., Gries D., Mulder M. C., Tucker A., Turner A. J. and Young P. R. Computing as a discipline. *Communications of the ACM*, 32, 1, 1989, 9-23.

[24] Creswell J. W. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.* Sage Publications, Thousand Oaks, California, United States, 2002.

[25] Dehnadi S. Testing Programming Aptitude. *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group.* Brighton, United Kingdom, 2006, 22-37.

[26] Dehnadi S. and Bornat R. The camel has two humps, 2006.

[27] Dickmann R. A. and Lockwood J. Computer personnel research group, 1966 survey of test use in computer personnel selection. *Proceedings of the Fourth SIGCPR Conference on Computer Personnel Research.* Los Angeles, California, United States. ACM Press, 1966, 15-25.

[28] Dijkstra E. W. The humble programmer. *Communications of the ACM, 15*, 10, 1972, 859-866.

[29] Engel G. and Roberts E. Computing Curricula 2001 Computer Science, Final Report, 2001, 240.

[30] Epstein H. T. The Fourth R or Why Johnny can't reason. http://www.brainstages.net/4thr.html, last accessed, 2 July 2007.

[31] European Union. ECTS – European Credit Transfer and Accumulation System. http://ec.europa.eu/education/programmes/socrates/ects/index_en.html, last accessed 2 July 2007.

[32] Ferguson R. D. Abstraction Anxiety: A Factor of Mathematics Anxiety. *Journal for Research in Mathematics Education*, 17, 2, 1986, 145-150.

[33] Frorer P., Manes M. and Hazzan O. Revealing the Faces of Abstraction. *International Journal of Computers for Mathematical Learning*, 2, 3, 1997, 217-228.

[34] Hagan D. and Markham S. Does it help to have some programming experience before beginning a computing degree program? *Proceedings of the 5th Annual ITiCSE Conference on Innovation and Technology in Computer Science Education*. Helsinki, Finland. ACM Press, 2000, 25-28.

[35] Hartmanis J. Turing Award lecture on computational complexity and the nature of computer science. *Communications of the ACM*, 37, 10, 1994, 37-43.

[36] Hazzan O. How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science. *Computer Science Education*, 13, 2, 2003, 95-122.

[37] Hazzan O. and Kramer J. ICSE Workshop on the Role of Abstraction in Software Engineering, 2008.

[38] Hudak M. A. and Anderson D. E. Formal Operations and Learning Style Predict Success in Statistics and Computer Science Courses. *Teaching of Psychology*, 17, 4, 1990, 231-234.

[39] Inhelder B. and Piaget J. *The growth of logical thinking from childhood to adolescence. An essay on the construction of formal operational structures.* Basic Books, New York, New York, United States, 1958.

[40] Inhelder B. and Piaget J. *De la logique de l'enfant à la logique de l'adolescent. Essai sur la construction des structures operatoires formelles*. Paris, 1955.

[41] Kolb D. A. *Learning Style Inventory: Technical Manual.* McBer, Boston, Massachusetts, United States, 1976.

[42] Kramer J. Is abstraction the key to computing? *Communications of the ACM*, 50, 4, 2007, 36-42.

[43] Kurtz B. L. Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education*. Kansas City, Missouri, United States. ACM Press, 1980, 110-117.

[44] Leeper R. R. and Silver J. L. Predicting success in a first programming course. *Proceedings of the Thirteenth SIGCSE Technical Symposium on Computer Science Education*. Indianapolis, Indiana, United States. ACM Press, 1982, 147-150.

[45] Machanick P. Teaching Java backwards. *Computers & Education*, 48, 3, 2007, 396-408.

[46] Machanick P. The abstraction-first approach to data abstraction and algorithms. *Computers & Education*, 31, 2, 1998, 135-150.

[47] McKinnon J. W. and Renner J. W. Are Colleges Concerned with Intellectual Development? *Am. J. Phys.*, 39, 9, 1971, 1047-1052.

[48] Nguyen D. Z. and Wong S. B. OOP in introductory CS: Better students through abstraction. *Proceedings of the Fifth Workshop on Pedagogies and Tools for Assimilating Object-Oriented Concepts*, OOPSLA '01, Tampa, Florida, United States, 2001.

[49] Nielsen H. and Thomsen P. V. *Hverdagsforestillinger i fysik (Everyday notations in physics*. In Danish). Gymnasiefysikrapport nr. 1. Department of Physics and Astronomy, University of Aarhus, Aarhus, Denmark, 1983.

[50] Or-Bach R. and Lavy I. Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bull*, 36, 2, 2004, 82-86.

[51] Pillay N. and Jugoo V. R. An investigation into student characteristics affecting novice programming performance. *SIGCSE Bull*, 37, 4, 2005, 107-110.

[52] Pritchard M. E. and Wilson G. S. Using emotional and social factors to predict student success. *Journal of College Student Development*, 44, 1, 2003, 18-28.

[53] Rauchas S., Rosman B., Konidaris G. and Sanders I. Language performance at high school and success in first year computer science. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. Houston, Texas, United States. ACM Press, New York, NY, USA, 2006, 398-402.

[54] Richardson F. C. and Suinn R. M. The Mathematics Anxiety Rating Scale: Psychometric Data. *Journal of Counselling Psychology*, 19, 6, 1972, 551-554.

[55] Roberge J. J. and Flexer B. K. Formal operational reasoning test. *Journal of General Psychology*, 106, 1982, 61-67.

[56] Rountree N., Rountree J., Robins A. and Hannah R. Interacting factors that predict success and failure in a CS1 course. *Working group reports from ITiCSE'04 on Innovation and Technology in Computer Science Education*. Leeds, United Kingdom. ACM Press, 2004, 101-104.

[57] Sanders, R. T. Jr. *Intellectual and psychosocial predictors of success in the college transition: A multiethnic study of freshman students on a predominantly white campus.* University of Illinois at Urbana-Champaign, 1998.

[58] SAT. Encyclopedia Britannica. http://www.britannica.com/, last accessed 2 July 2008.

[59] Shayer M. and Adey P. *Towards a science of science teaching. cognitive development and curriculum demand.* Heinemann Educational, Oxford, United Kingdom, 1981.

[60] Simpson D. Psychological testing in computing staff selection: a bibliography. *SIGCPR Comput. Pers.*, 4, 1-2, 1973, 2-5.

[61] Soper D. S. A-priori Sample Size Calculator (multiple regression) *Free Statistics Calculators* (online software). 2007.

[62] Sprague P. and Schahczenski C. Abstraction the key to CS1. *J. Comput. Small Coll.*, 17, 3, 2002, 211-218.

[63] Szulecka T. K., Springett N. R. and de Pauw, K. W. General health, psychiatric vulnerability and withdrawal from university in first-year undergraduates. *British Journal of Guidance & Counselling Special Issue: Counselling and Health*, 15, 1987, 82-91.

[64] Thomas L., Ratcliffe M., Woodbury J. and Jarman E. Learning styles and performance in the introductory programming sequence. *SIGCSE Bull*, 34, 1, 2002, 33-37.

[65] Ting S. R. and Robinson T. L. First-year academic success: A prediction combining cognitive and psychosocial variables for Caucasian and African American students. *Journal of College Students Development*, 39, 1998, 599-610.

[66] Turner A. J. Computing Curricula 1991. *Communications of the ACM*, 34, 6, 1991, 68-84.

[67] Ventura P. R. On the origins of programmers: Identifying predictors of success for an objects first CS1. PhD Thesis, The State University of New York at Buffalo, USA, 2003.

[68] Ventura P. Identifying predictors of success for an objects-first CS1. *Computer Science Education*, 15, 3, 2005, 223-243.