To Project or Not to Project: In Search of the Pathway to Object Orientation

Arno Pasternak[1] and Johannes Fischer[2]

[1] Fritz-Steinhoff-Schule Hagen arno.pasternak@cs.tu-dortmund.de
[2] Technische Universität Dortmund johannes.fischer@cs.tu-dortmund.de

Abstract. We study two different ways to learn object-oriented programming in higher secondary education:
The first way is to structure the lessons around an overarching project. First, a smaller object-oriented program is introduced. During the learning process, this program and the associated tasks become more and more complex and difficult in structure and size. In the second way, the individual lessons are structured around various small modules, each introducing new aspects. We call this approach LULUM (Learning Using Little Useful Modules). We evaluated the approaches with two groups of students from a comprehensive school. In addition, we compared these results with those of another group from a grammar school.
Our results are as follows:
When comparing the two approaches, the LULUM approach has shown to be more successful. The question nevertheless arises as to whether it makes sense to lay a relatively solid foundation in imperative programming before introducing the fundamentals of object orientation.

Keywords: Higher secondary education, Object Orientation, Project, LULUM, Modules, CS for all

1 Introduction

Computational Thinking[15] consisting of the key concepts of *abstraction* and *automation*, has been identified as one of the human ways of thinking and provides essential insights similar to e.g. mathematical, physical and social scientific thinking.

The great power of CS is to transfer any kind of problem with its possible solution to the machine level by an abstraction process and thus enables it to be executed. This is known to be a difficult process, as it has to combine the steps *modeling* and *programming*.

1.1 Short historical Overview of the Development of Programming languages

Two different paths to programming can be identified in the history of CS:

One path has been taken from the machine level to the problem level. Since programming at the machine level with its very simple memory structures for numbers and characters was too intransparent and bug-prone, the development of ever more complex languages moved further away from the machine level. In the beginning, the focus was primarily on the formulation of processes and less on data, whose structure was still very close to the machine due to the mostly mathematical problems. In a second stage of development these data structures became more and more complex. These languages mostly belong to the group of *imperative languages* [14,11,12]. In school, programming as well as modelling is primarily done according to the imperative paradigm. If at all, model computers were realized according to this concept.

The second way started from the mathematical description of problems. The concrete realization on a machine was actually not so important. The result was *declarative programming*. In terms of data structure, the list was the only built-in structure. All other data structures can be represented by this structure [14,11,10]. CS in school did not follow this path. Lists were introduced into the school curriculum exclusively as variables with dynamically allocated memory in imperative programming languages.

The concept of *object-oriented programming* takes a special role. The expansion and, at the same time, the restriction compared to already existing modular ideas consists of the reduction of modularity to data capsules, generally called *classes*. These were first introduced in the *Simula* language [4,3] as an extension of the *record types* in languages such as *Pascal*. As a language-defining construct, this idea has been realized in *Smalltalk*. This language reduces all programming to the *Object Message Principle* [8]. The process control thus becomes part of the data structuring. In this sense, the programming paradigm can then be called its own [14,4,3,8,11]. This approach has *not* been accepted in mainstream languages, despite assertions to the opposite. In other languages, object orientation has been added to the data structuring that was common until then and has not changed the mostly imperative flow control. The gain of such an object-oriented view is then primarily a profit when designing large systems, and no longer as a programming paradigm. Most of today's so-called object-oriented languages are therefore essentially imperative languages. In school, the contents of imperative modeling and programming have been transferred to object-oriented description on an imperative basis.

1.2 Educational Consequences of OOP

The more extensive and complex the structures of a programming language are, the more complex its syntax tends to become. For the learnes, this means that they have to master many ideas at a very early stage when being introduced to these languages. For example, the use of classes as a special modular concept requires a broad understanding of procedures and functions, including the concept of return values.

From a didactic point of view, this requires careful reflection on how and when the concepts of modeling and programming should be taught [6]. In the

past, teaching was mainly limited to the advanced levels of the school education and therefore the teaching of these structures was limited to a very short period of time. We know that teaching subjects are more effective if they are spirally processed several times over a number of years. Regarding the teaching of OOM and OOP, there is hardly any experience in this area [1].

In the future, teaching CS will be increasingly integrated into the lower grades of the school and in some circumstances extended to the primary level. Therefore it makes sense to think more about when which concept should be introduced in the classroom.

1.3 Object Oriented Modeling and Programming in School

Since most of the lessons are currently taught in the higher secondary levels, we limit ourselves to these grades. The aim is to introduce the basic ideas of OOM and OOP. Currently this is mostly done in the programming language *Java* or in languages similar to Java.

Of course, formulating the goals and choosing a programming language is not sufficient. We need both didactically and methodically well justified procedures. Of course, we do not want to teach a programming language reducing to the syntax of a programming manual, but prefer to do it with examples, which, if possible, should be taken from real life contexts, to make the students familiar with the subject. A central didactical question here is whether it makes sense to do this in a scenario typical to the work of a computer scientist, i.e. a project, or whether it is better to choose smaller examples according to the individual concepts to be taught, so as not to overburden the students.

1.4 Research Questions

1. Are there significant differences in learning outcomes between project-based learning and learning with more appropriate examples? Therefore we examined two comparable groups at a comprehensive school. We limit ourselves to relatively small projects, as they were shown to be more successful in an earlier study [5].
2. Are there differences in the increase in learning compared with established subjects? Thus at the end of the survey, we additionally examined our CS students on their mathematical knowledge on fundamental facts.
3. Are there differences between grammar school and comprehensive school in respect of learning success? We surveyed students at a grammar school to find out how strongly the specific conditions at the examined comprehensive school influence learning success in CS.

2 Teaching Framework

In Germany CS is not a compulsory subject, but it is an elective subject at many schools in higher secondary education. In many cases, the courses are attended

by students who have a rather strong interest in computer science. However, our goal is that all students are taught in CS. This will certainly have an impact on the overall pace of teaching and additionally on the curriculum. This requires the practical implementation of the 'Computer Science Teachers Association' (CSTA) from the year 2016: 'It is intended to introduce the principles and methodologies of CS to all students, whether they are college bound or career bound after high school' [2, p.7]. Thus the principles of *Computational Thinking* [15] are implemented.

The following characteristics apply to the school in which one of the authors teaches: Almost half of the students choose computer science, of which again almost half are female. Almost $\frac{3}{4}$ of the students have a migration background. For most students CS is a 'normal' subject, which means that they do not spend more time on CS than on other subjects. Students spend very little time doing their homework.

We chose two different approaches for teaching object-oriented modeling and programming (OOM and OOP), as outlined next.

2.1 Approach 1: Small Projects

Sentance and *Waite* describe an approach called *PRIMM* [13], which consists of the steps *Predict, Run, Investigate, Modify, Make*. Some of their ideas were borrowed from *Kölling* and *Rosenberg* [9], but they work with (relatively) large projects, whereas we preferred small ones.

This group started with a project with about 65 lines of code, which simulated a card game. The game initially consisted of a `stack_class`, modeling a stack of cards. The individual cards had a unique number as value. In the actual application, two stacks were created, and cards were drawn alternately from the stacks and checked.

The basic actions are: The program was first read and analyzed. Then small modifications and later extensions of the class were made. The target situation of the card game was: Two players of the `players_class` play a simple card game. Each has a deck of cards of the `stack_class`, which consists of a fixed number of cards from the `playing_card_class`.

2.2 Approach 2: Learning using little useful modules (LULUM)

In this approach the concepts were introduced with small modules in form of classes. If possible, the modules were taken from the students' real life. However, the contexts differed. Otherwise, the methodological procedure did not differ from that in approach 1.

In this second group, a class for the representation of fractions was introduced as a first example. Another smaller example realized an exchange office for different currencies. Afterwards a class for the processing of accounts with passwords was edited. Since the context was changed in each case, it was always possible to work with shorter program texts.

2.3 Group Structure

In the 2018/2019 school year two parallel groups were set up. Both groups were almost the same number of students, 20 and 19 respectively. The split of the students into the two courses was made for administrative reasons and therefore practically randomly with concern to the knowledge in computer science. However, the proportion of female students was, at one third, smaller than in recent years, but still significantly higher than at other equivalent schools.

All students had lessons with another teacher in the previous school year. They had no CS teaching before in the lower secondary education. Therefore, to get to known each other at the beginning of the school year, a teaching unit with an introduction to *LaTeX* was held. Subsequently, the (imperative) programming with the data structure *array* introduced in the previous school year was repeated before the introduction to object-oriented programming according to one of the approaches described above was started. This introductory unit had a length of 10 teaching weeks in both courses with three hours per week each.

3 Description of Empirical Research

3.1 Description of the Questionary

We asked at several points for a description of the terms *field, class, attribute, method, object, abstract class, abstract data type, OO modeling and programming*: before the series of lessons (pre), at the end of the 10-weeks unit (post), and after another 6 weeks (long-term) . In addition, everytime we asked for the modeling of a `car_class` for a car shop. So, in essence, only reproductive knowledge on the lower levels of learning taxonomies were necessary.

The students' statements were translated into a numerical value of a *Likert scale* from 1 to 6 for the assessment *No answer, wrong, mostly wrong or incomplete, partly wrong / partly right, mostly right, right*.

For a reasonable evaluation of the results we evaluated the *effect size* d according to *Hattie* [7] for the individual subquestions. In this way we cannot only measure the results of the two groups, but also whether a successful learning effect can be observed at all. According to Hattie, the goal is to achieve an effect size of $d \geq 0.4$ in a one-year course of about four hours per week. Since our unit took only a few weeks, even correspondingly smaller values are of interest.

3.2 Differences between the two Teaching Approaches

We investigated the two groups *Project* and *LULUM*. Let us first look at the learning outcomes of the students.

We can recognize in Fig. 1 that almost all students are not familiar with the terms used in object-oriented programming at the beginning of the lesson. Fig. 2 shows that in addition to the students who have made only little progress, some students did indeed progress. It can also be seen that the learning increase of

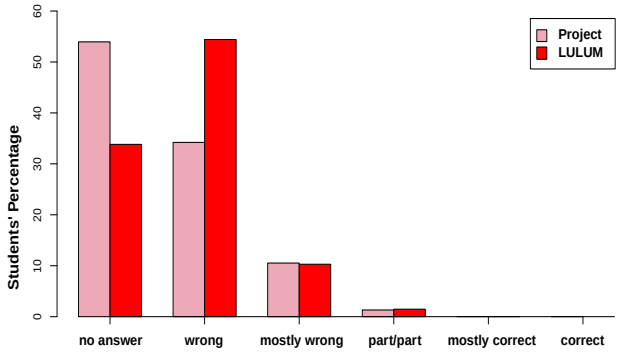

Fig. 1. Pre Answers (before teaching unit): Project/LULUM

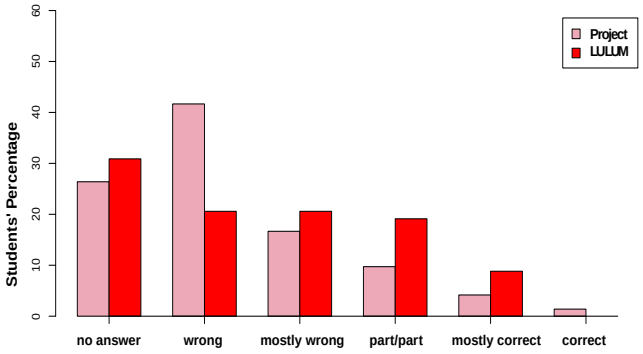

Fig. 2. Post Answers (after 10 weeks): Project/LULUM

the students in the *LULUM* group is slightly above that of the project group. The question arises whether these results are statistically significant.

3.3 Evaluation of Selected Statistical Results

An evaluation of the effectiveness of an educationally relevant measure can be made with the *effect size* according to *Hattie* [7, p.8]. With this metric, different groups can be compared on a linear scale with regard to a measure.

Effect size	**Total**		class	operation	object	car class
Project	**0.48**		1.51	0.28	0.33	0.74
LULUM	**0.80**		2.13	1.24	0.78	0.54

Table 1. **Effect size d for the different groups.** d describes the increase in performance in each of the different groups from the beginning (pre) to the end (post) of the intervention.

Table 1 shows the learning effects (individually for the two groups) for the four test items between the beginning (pre test) and the end of this unit (post test) introducing OOM and OOP, and also a summary over the four items ('Total'). It shows a significantly better learning effect ($d_{diff} = 0.8 - 0.48 = 0.32$) in favour of the LULUM group compared to the project group.

A second variant for measuring the *effect size* is to perform a measurement between two groups. Table 2 shows the differences between the two groups at the beginning and end of the lecture.

The LULUM group is better in almost all values. The better value in the pre-post scoring for developing a `car-class` is due to the fact that except for

Effect size	**Total**		class	operation	object	car class
pre	**0.21**		0.35	-0.25	0.48	0.73
post	**0.40**		0.39	-0.29	0.66	0.18

Table 2. **Effect size d comparing Project and LULUM.** d describes the difference in performance between the two groups at the beginning (pre) and at the end (post) of the intervention.

two students in the project group, who wrote nothing at all in the initial test, all the students in the LULUM group tried a little to answer, which was then mostly completely wrong. Otherwise, the LULUM group signs a more significant learning increase than the project group.

We can observe that the effect size for the pre- and the post-examination between the two groups increases from $d = 0.21$ up to $d = 0.40$. This shows also here a clearly better learning effect ($d_{diff} = 0.40 - 0.21 = 0.19$) in favour of the LULUM group compared to the project group at the end of the teaching unit. This value is strongly influenced by the many wrong, mostly wrong and missing answers. If we ignore the learning progress within the lower levels from 'no answer' to 'mostly wrong' and limit it to those students whose answers are at least partially correct, we still get an effect size of $d = 0.27$ in comparison to $d = 0.40$ by all students in favor of the LULUM group.

Unfortunately, there is a non-negligible number of students who have experienced practically no learning increase. We want to know how the learning effect of students who really show an increase in learning has developed. If we take the liberty to exclude five of these students from each group, the learning gain of the project group is $d = 0.52$ in direct comparison to $d = 1.00$ for the LULUM group. The effect size of the LULUM group compared to the project group increases from $d = 0.27$ at the beginning to $d = 0.50$ at the end of the lesson.

Overall we can answer research question 1 to the extent that there is a significant difference in learning outcomes in favour of the LULUM group.

3.4 Other Interesting Descriptive Statistical Results

Results at the end of the activity To evaluate long term learning effects, Fig. 3 and Fig. 4 show the results of the final test conducted 6 weeks after the teaching unit. The results are not satisfying at all: overall, more than 50% of the students' comments on all questions were wrong, or the students did not answer at all. Another approximately 15% students answered overwhelmingly wrong. It is noteworthy that this contradicted the teacher's feeling during the lesson and when observing students' activities on the computer.

It is also surprising that the most basic concept (*class*) shows even worse results than concepts that build on it (*operation* and *object*). Obviously, the fundamental idea of describing similar objects in a logical structure has not really been understood.

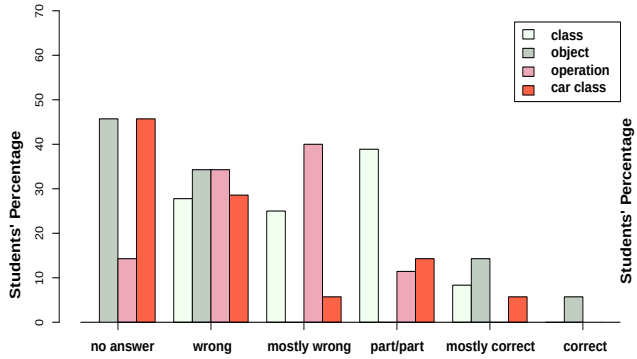

Fig. 3. Post Answers about the Items
class, operation, object, example carclass

Fig. 4. Post CS Answers Summary

A problematic situation arose from the fact that students in the lower secondary education have no teaching experience in CS. Is the rate and quantity of learning material generally too high, so that no permanent effects are achieved for half of the students? Is the teaching in CS generally ineffective? Or does this result have nothing to do with the subject CS?

Comparison CS with Math We verified this by asking the students again about 2 months after this post-survey, and additionally asked them for comparable mathematical terms – function, indefinite integral, definite integral, example in/outflow of a sea – to examine their mathematical understanding and knowledge at a similar level.

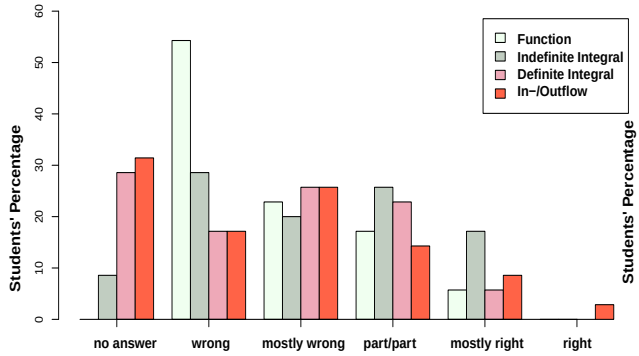

Fig. 5. Math Answers about the Items
function, indefinite integral, definite integral, example in/outflow

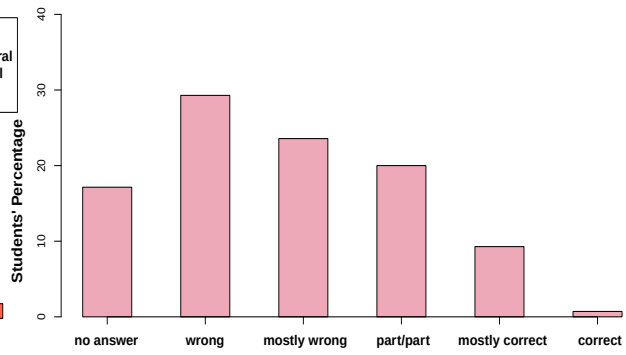

Fig. 6. Math Answers Summary

We did not expect any mathematical excellence in this supplementary survey, but the results according to Fig. 5 and Fig. 6 are also disillusioning in the end. Also in Mathematics taught by various teachers, almost half of the students show that they did not achieve the mathematical goals. This is surprising, given that students have been taking math as a subject since they started school more than 10 years ago. The concept of *function* has been spiralled through more than five years and still more than 50% of the students cannot sufficiently explain this concept.

Fig. 7 compares the results and we get the answer to research question 2: Hardly any difference can be observed between CS and Mathematics: about half of the students have learned nearly nothing in both subjects. These results in both Mathematics and CS cannot satisfy students, teachers and above all society. It is to be feared that this is also true in a similar way for at least some of the other subjects.

If these results are transferable to the majority of students and subjects, it is necessary to identify which topics and problems should be offered in many subjects so that most students show a greater learning effect than at present.

Comparison of Students from Different School Systems The question arises whether this basic behaviour applies generally to all schools. Many parents try to enroll their children directly at the *grammar school*, so that the comprehensive school often lacks the intellectually more capable students. This could explain why the standards that apply to all schools are only fulfilled by a part of the students at our school.

In order to examine this, parallel to the survey at the examined comprehensive school we also interviewed students at a Gymnasium who had taken CS as a school subject there. These results are not fully comparable, as we do not have an overview of which subjects were taught at this school and how intensively. However, the same curriculum applies to all students and teachers. The composition of the courses at the grammar School compared to the comprehensive school is not entirely comparable. Both the percentage of girls and the percentage of students with foreign background is more than twice as high at the comprehensive school.

We can see in Fig. 8 that in both Mathematics and CS the students' performance is significantly better than in the comprehensive school. However, even here more than 40% of the students remain in the unsatisfactory range. Since the surveyed grammar school also has an advanced placement course (AP) in CS, the figures in CS are not quite adequate.

If one looks only at the base course students in CS in Fig. 9, it becomes clear that the performance in CS is also worse: more than 50% of the students do not achieve sufficient performance, either.

In summary, research question 3 can be answered that students tend to do better at grammar school. However, if only the students in the basic courses are taken into study, the results come closer.

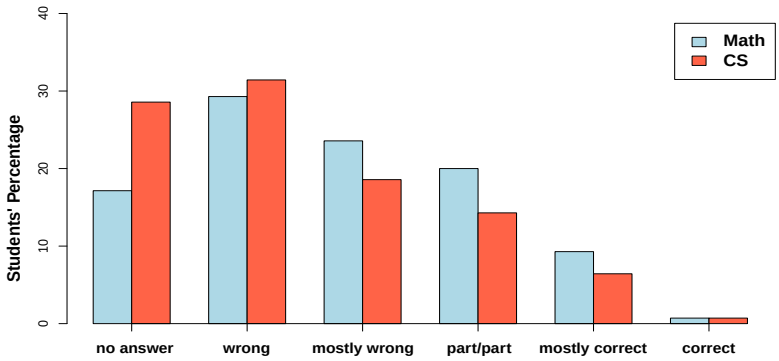

Fig. 7. Comparing Mathematics with CS (comprehensive school)

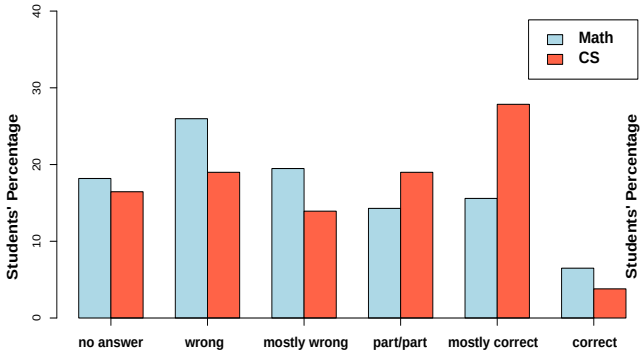

Fig. 8. Math/CS answers (gram. school)

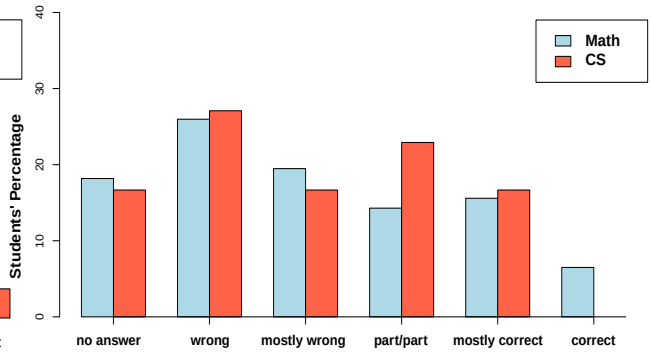

Fig. 9. Math/CS (gram. school w/o AP)

3.5 Interpretation of the Statistical Results

At the beginning of the investigation, about 90% of the students were not able to answer the questions about object-oriented programming correctly. This percentage drops to about 60% in the project group and just over 40% in the LULUM group. There is no doubt that the students have learned quite a bit overall. However, about half of the students have learned practically nothing. The questions were largely reproductive in nature. The only non-reproductive question was answered even worse.

The approach for both groups was object-oriented programming. The goals were the same, only the didactic focus was different. The claim of a general education school for all students is to teach essential ideas in such a way that

the majority is able to understand the ideas to be taught, at least in principle. This has obviously not been fulfilled.

A possible explanation is that the idea behind these lessons is too difficult for a large part of the students at this time with their personal learning background.

It cannot be denied that the relationship between the modeling level and the implementation level in object-oriented modeling and programming is only comprehensible to those who already have considerable experience in programming.

If this is correct, this learning process must be preceded by learning exactly these structures in whatever form in the context of a spiral curriculum in the sense of the ideas of Bruner [1].

Comparing the results of the survey at the comprehensive school with those at the grammar school, this interpretation is substantiated. There, too, it can be recognized that for a large proportion of the students, especially those who do not attend a more in-depth education in CS, the object-oriented approach does not produce the desired outcomes.

At the comprehensive school it is also shown that students who have got to know different aspects of object-oriented modelling and programming with different smaller examples (LULUM) have a significantly higher learning effect. This is true for the overall group as well as the limitation to those students who have at least partially understood the ideas correctly.

4 Summary

School has the task of conveying the essential ways of thinking. Apart from, for example, mathematical, physical and social-scientific thinking, this also includes *Computational Thinking*. According to Jeannette Wing this consists of *abstraction and automation* [15]. Abstraction is implemented by *modeling*, automation by *programming*. At the modeling level, the world is represented by *abstract data types*, which are modified by recursive and/or iterative flow structures. *Object-oriented modeling and programming OOM/P* has become widely accepted as a concrete technology in industrial practice.

A dominating position in didactics is to practice this approach in schools as well. Due to the complexity of the programming language, it is therefore important to determine how this is practised in school. Therefore, we examined whether working with small projects or with several independent examples (LULUM) is more suitable.

It can be seen across all students that the assumption that the view of object orientation, which is a very appropriate approach from the perspective of professional computer science, causes more difficulties for students the less they are involved in the subject CS. It is impossible to determine whether this is caused by the inherent difficulties of object orientation, or by the high demands of the syntax of object-oriented languages.

However, the proportion of students who did experience learning growth is significantly larger in the LULUM group. We only examined two groups of about 20 students each. If these results are also valid for relatively large groups

of students, a rethinking of CS didactics is necessary. Maybe in a *CS for All* curriculum programming should not start with the OOM and OOP, but at much easier requirements? Are CSUnplugged teaching units helpful in this respect? And what is the influence of the chosen programming language.

References

1. Bruner, J.S.: The process of education. Harvard University Press (1960)
2. CSTA STANDARDS TASK FORCE: [Interim] CSTA K–12 Computer Science Standards: Revised 2016. Tech. rep., New York, NY, USA (2016)
3. Dahl, O.J., Myhrhaug, B., Nygaard, K.: Some features of the simula 67 language. In: Proceedings of the Second Conference on Applications of Simulations. pp. 29–31. Winter Simulation Conference (1968)
4. Dahl, O.J., Nygaard, K.: Simula: An algol-based simulation language. Commun. ACM **9**, 67–78 (1966)
5. Fischer, J., Pasternak, A.: Comparing approaches for learning abstraction and automation by object orientation. In: Jasutė, E., Pozdniakov, S. (eds.) 12th International conference on informatics in schools * Situation, evaluation and perspectives. pp. 39–47. University of Cyprus, Larnaca (2019)
6. Flores, P., Torres, J., Fonseca-Delgado, R.: Design decisions under object-oriented approach: A thematic analysis from the abstraction point of view. pp. 89–97 (11 2019)
7. Hattie, J.: Visible Learning: A Synthesis of Over 800 Meta-Analyses Relating to Achievement. Taylor & Francis (2008)
8. Kay, A.C.: The early history of smalltalk. SIGPLAN Not. **28**(3), 69–95 (1993)
9. Kölling, M., Rosenberg, J.: Guidelines for teaching object orientation with java. In: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education. pp. 33–36. ITiCSE '01, ACM, New York, NY, USA (2001)
10. Queinnec, C.: Lisp in Small Pieces. Cambridge University Press (1996). https://doi.org/10.1017/CBO9781139172974
11. Sammet, J.E. (ed.): HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages. Association for Computing Machinery, New York, NY, USA (1993)
12. Sammet, J.E. (ed.): HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. Association for Computing Machinery, New York, NY, USA (2007)
13. Sentance, S., Waite, J.: Primm: Exploring pedagogical approaches for teaching text-based programming in school. In: Proceedings of the 12th Workshop on Primary and Secondary Computing Education. pp. 113–114. WiPSCE '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3137065.3137084, http://doi.acm.org/10.1145/3137065.3137084
14. Wexelblat, R.: History of Programming Languages. Elsevier Science (2014)
15. Wing, J.M.: Computational thinking. Communications of the ACM **49**(3), 33–35 (2006)