Chapter # 42

REFLECTIONS ON DIDACTICAL CHALLENGES IN TEACHING COMPUTER PROGRAMMING

Marcin Fojcik[1], Martyna K. Fojcik[2], Sven-Olai Høyland[3], & Jon Øivind Hoem[4]
[1]*Department of Electrical Engineering, Western Norwegian University of Applied Sciences, Norway*
[2]*Faculty of Humanities and Education, Volda University College, Norway*
[3]*Department of Electrical Engineering, Western Norwegian University of Applied Sciences, Norway*
[4]*Department of Arts Education, Western Norwegian University of Applied Sciences, Norway*

ABSTRACT
One of the key challenges of modern society is the vast number of technological devices surrounding us. As a result, general ICT skills are essential for both work and personal time. In addition, ICT skills are widely used in the education of different subjects. The challenge is that while computer science (programming) is relatively well covered in the literature, computer science in other professions, including education for non-IT professionals, is not.
Teaching computer programming is particularly difficult in courses for students from outside computer science fields. The fundamental problem is: what computer skills should be taught and to what extent? It is usually impossible to teach all possible concepts in a course. In this case, the focus should be on programming terminology, key definitions, or perhaps - computational thinking and problem-solving skills? Another challenge may be using English or the national language and selecting examples based mainly on mathematics or practical experiences.
This chapter presents the experiences and reflections of authors from different universities, departments, and courses on teaching, using other pedagogical approaches, and programming theories comparing programming for computer science students and non-computer science students.

Keywords: programming skills, digital competencies, 21st-century skills, didactics in IT education, introduction to programming.

1. INTRODUCTION

Technology has never before been developed as quickly and is quickly becoming outdated. Technological development has recently increased digital literacy in industry, education, business, research, and personal lives. Digital Literacy is a collection of digital competence, digital usages, and digital transformation, which can be defined as:

> *the awareness, attitude, and ability of individuals to appropriately use digital tools and facilities to identify, access, manage, integrate, evaluate, analyze and synthesize digital resources, construct new knowledge, create media expressions, and communicate with others, in the context of specific life situations, in order to enable constructive social action; and to reflect upon this process* (Martin, 2006, p.155).

This needs for skills and literacy towards digital development has caused many students to learn to program. As a result, programming has been introduced to other fields than computer science and information technology. This chapter aims to provide insight into how it can be done successfully and shed light on the challenges in teaching programming for non-programmers. This chapter tries to present some of the most common challenges (from different researchers) and match them to the pedagogical concept of the Didactical

Triangle proposed by Kansanen & Meri (1999). The research question discussed in this study is: What can be challenging in introducing programming based on the Didactical Triangle: Content, Teacher, and Student?

A discussion on challenges in teaching programming needs to address what programming is and that this term has a broad understanding. This chapter is based on Hartree's (1950) definition: "The process of preparing a calculation for a machine can be broken down into two parts, 'programming' and 'coding'. Programming is the process of drawing up the schedule of the sequences of individual operations required to carry out the calculation" (cited in Blackwell, 2002, p. 204). This definition differentiates between writing the code (coding) in a chosen programming language and designing a program through "planning, scheduling and performing a program" (merriam-webster dictionary, 2020). These actions were separate at the start of the computer era, but as the elements and methods in programming changed, there was a need for different strategies and structures.

Today many people mix the terms coding and programming. Blackwell states, "when people say they are programming, we should not question whether this activity is genuine programming, but instead analyze their experiences to understand the general nature of programming activity" (Blackwell, 2002, p. 208). Some of the findings of Schulte and Bennedsen (2006) suggest that many students view programming as coding in a typical introductory programming course because the teachers tend to focus on concrete details like notations rather than general understanding and structure.

This theoretical and reflective chapter is divided into five sections. The "*Background"* shows issues related to the dynamic expansion of IT and ICT fields and its educational challenges. Next, *"Challenges in Practice"* presents programming education challenges and some central aspects concerning the didactical triangle: Content, Teacher, and Student (Kansanen & Meri, 1999). Then the "*Discussion"* shows some thoughts and comments on differences and solutions. The last part, "*Conclusions",* gathers presented theories and examples in the conclusion.

2. BACKGROUND - ABOUT LEARNING AND TEACHING

Today, IT is changing more than any other field due to its integration and collaboration with other disciplines. There are continuously new elements and modifications to the existing ones. Previously used, considered the correct way of working, is obsolete and no longer meets the requirements (time of preparation of the application and its security). Changes in programming methods correspond to changes in the ways components are connected and used in the industry. Nowadays, it is not enough to know one programming language to have knowledge about writing programs. It is necessary to know the whole process, containing many elements, structures, methods, networks, environment, and security. That involves much more than just a language. The program must have a modern graphical interface, the ability to communicate with modern networks (communication protocols), multilingualism, and the possibility of easy modification and development. Looking for a job by a person with basic programming knowledge (Java, JavaScript, Python, C++, C#) without knowledge of the environment is a misunderstanding of the requirements. Such a person is not a professional candidate for the job.

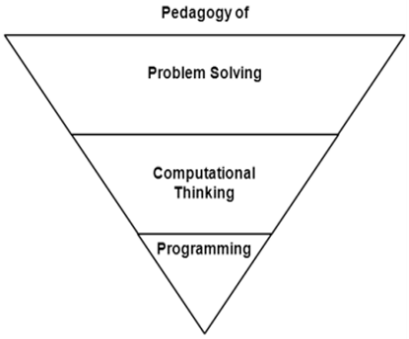
Programming itself is considered hard to learn (Sheard & Carbone, 2007). Different programming languages, technical differences, teaching methods, and personal approaches can affect the learning of programming (Demir, 2022; Robins, Rountree, & Rountree, 2003). In addition, there is also the gap between standardized education and the continuously changing requirements of employers. The issues lie with the use of the equipment, naming issues, language issues, methodologies, and structure of the task. Even the way of teaching

can present a different world in a lecture, especially regarding the definition from various fields (Blackwell, 2002). The result is that some students choose to quit their course because they misunderstand a subject's requirements or other expectations.

One of the reasons for this gap between education and industry is rapid technological development. While education is focused on basic structures and standard terminology, the industry is taking the next step in introducing ideas like the Internet of Things, Factory 4.0, Artificial intelligence, and Big Data. These concepts are similar in the way technology connects different aspects of technology. Still, they are used in various environments with different requirements, both for hardware and software. According to Bettin, Jarvie-Eggart, Steelman, and Wallace (2022, p. 309): "[u]nderstanding the static structure of computer programs and understanding the dynamic structure of program development are both vital competencies for novice programmers". So why is education not teaching how to implement those ideas? How is education prepared for these challenges? Do university students have the opportunity to obtain the necessary knowledge to the extent employers require? What skills are essential to living in the modern computerized world?

Fojcik, Galek, and Fojcik (2017) show in their research that many students do not have sufficient skills to evaluate their digital literacy. These students believe they have efficient ICT knowledge if they are fluent in using everyday technological equipment like smartphones, computers, social media, or software. Therefore, there is a need for more education about what requirements are necessary to be up to date with ICT knowledge and digital literacy (Vuorikari, Punie, Carretero, & Van den Brande, 2016). There is no doubt that society needs to know more about technological tools than "where to push the button", but the response to this need should not be introducing the newest ideas like machine learning or artificial intelligence. Higher education cannot fully implement theories and concepts still in development.

Figure 1.
Conceptual framework (Selby, 2014, p.19).

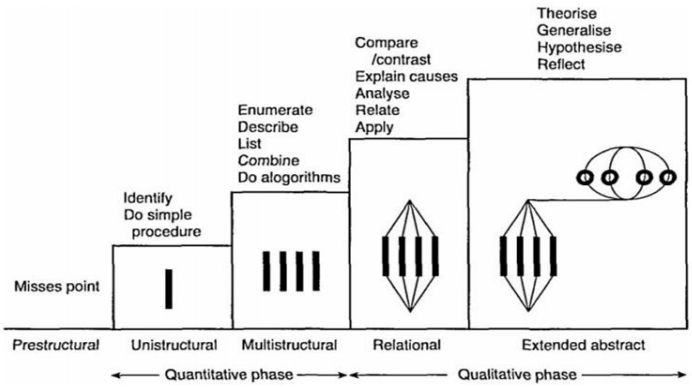


To cope with the challenge, some researchers present a set of best practices in teaching programming (Brown & Wilson, 2018, Cawthorne, 2021). Hence, the standard of university education delivers quality, research-based knowledge, and continuous change to modern ideas (that may not work, lose their popularity, or be exchanged for another idea) would not preserve the standard. A possible solution for filling this gap can be to teach skill development. Robins et al. (2003) proposed more explicit attention to programming strategies when introducing programming to students, and Demir (2022) suggested that there should be an integration between the theory and practice of programming to have a positive effect on learning. Griffin and Care (2015) divide the "hard skills" like knowing a programming language, theory of regulation, mathematical operations, etc., and "soft skills"

like time management, creativity, collaboration, critical thinking, etc. Showing that to better fill the industry's requirements, the students should learn both the hard and soft skills so that they can adapt to the technical specifications that will be present when they finish their education. To be a professional programmer, you need to know how to plan, do task analysis, use computational thinking, create algorithms, structure tasks, and know some programming languages, coding, and programming environment. Most of these elements are elements common to problem-solving strategies. This can be seen in figure 1.

There are different approaches to introducing and implementing programming for students. One necessary element is assessing students' knowledge and skills. According to De Raadt, Watson, and Toleman (2009, p. 54) "[s]tudents seem to learn and apply programming strategies more consistently when they are presented in an explicit manner than when they are learned implicitly". By knowing what students can do and what they already know, the teacher may prepare the required proregression in the topic or prepare repetition of concepts that are not fully understood. A SOLO-taxonomy (Structure of the Observed Learning Outcomes) may help analyze student knowledge (Biggs, 2012) (figure 2). Studies that have used SOLO-taxonomy to assess programming courses (for example, Fojcik et al., 2020; Ginat, & Menashe, 2015; Malik, Tawafak, and Shakir, 2021; Sheard et al., 2008) show that it can be challenging for students to reach a higher level of observed learning outcome, and as Winslow stated "[t]he key to reaching this level is practice, practices and practice – starting with simple facts and problems and working up to more complicated facts, strategies and problems" (1996, p. 21).

Figure 2.
A hierarchy of verbs used in forms of curriculum objectives (Biggs, 2012, p.48).



The assessment of programming skills can be formative or summative. It can be written, oral, project-based, presentation, or practical exercises. Different learning approaches prioritize different working environments and challenge students in different ways. For example, writing a program may show if a student is able to plan and prepare the necessary structures, doing a project can present if a student can collaborate and implement working methods on a larger scale, an oral exam where a student is presenting their knowledge on a topic can assess of the student can use and exemplify the theoretical concepts with practical applications.

For example, a teacher learning programming must learn the programming language, working methods, and structures. Still, they also need to be able to explain this knowledge to others (as teachers do). In contrast, an automation engineer does not need to be great at explaining because they must connect the machines and supervise the program in all security aspects.

Diagram 1.
Description of how well the course from Volda University College, fits the IT and teacher criteria of SOLO-taxonomy. (Fojcik et al. (2020)).

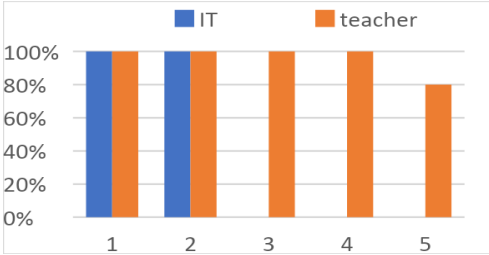
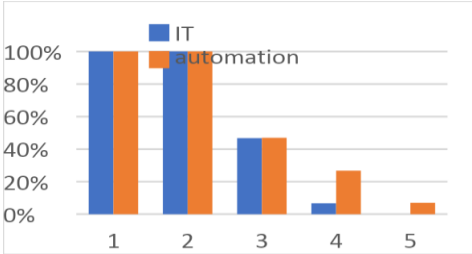


Diagram 2.
Description of how well the course from Western Norway University of Applied Sciences fits the IT and automation criteria of SOLO-taxonomy. (Fojcik et al. (2020)).

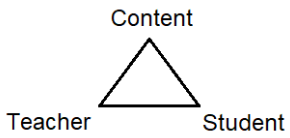


Fojcik et al. (2020) show that assessing non-computer programming students to "pure" programming or general IT programming (marked blue on Diagram 1 and Diagram 2) does not fill all the required learning outcomes, but if the criteria for assessment changes from general IT programming, to the specific needs of the programming outcomes in their field, for example in teacher education, the level of observed learning outcomes (figure 2) have increased (marked orange on Diagram 1 and Diagram 2).

3. CHALLENGES IN PRACTICE

Programming is not an easy course (Blackwell, 2002). There are challenges in teaching. For example, presenting information clearly and interestingly, motivating the students, keeping a preferred speed, assessing students, giving students feedback, and many more. In addition to the issues in teaching, some challenges appear when teaching programming. To understand the challenges in teaching programming, one must understand the didactics of programming. Kansanen & Meri (1999) present programming in the Didactical Triangle, with communication between a teacher, a student, and a content, where there is special attention to the context of a learned situation that affects the teacher and the student (figure 3). This part of the chapter will consider every element of the Didactical Triangle to present some opportunities and challenges noticed during the literature analysis supported by the authors' experiences. These elements are related to independent phenomena, but they can overlap while teaching.

Figure 3.
Model based on Didactical Triangle (Kansanen & Meri, 1999).



3.1. Content

3.1.1. Anthropomorphizing metaphor

In teaching programming, the content is central. To program is to create new solutions to problems using machines. A challenge that may arise is understanding the difference between tasks that are best solved by humans and tasks that machines can do better.

The anthropomorphizing metaphor can hinder learning programming (Dijkstra, 1989). This is a misleading phenomenon where the machine is treated like a human, not in feelings and opinions, but in decision-making. Students who look at a computer as a judge or an assessor do not understand the process of programming and designing precise instruction. It is a common mistake to say that "a program" is wrong. The only mistake a program can have it's the programmer's mistake. The program just follows the instructions and does not influence the algorithm in any way. So, if there is something wrong, it means that the program was not written correctly.

3.1.2. Different approaches

Programming has its set of rules and structures that a program needs to follow as a field of knowledge. However, there are still many options for scheduling and building a program. One approach is to look at procedural programming. When the programmer gets several new features and writes a program, he can use the procedural method to enrich the code from what they have previously learned. This approach builds student knowledge step by step, always returning to the presented facts, even if they might share a narrow perspective on the new topic (Berglund & Lister, 2010). Another approach is looking at the new topic's most significant advantages (and differences), presenting the ideas first, and then going back slowly to recap how the new topic fits the rest. (Berglund & Lister, 2010).

3.1.3. Specialized content for the course

As previously mentioned, the content in teaching programming can vary from the course it is presented in. For example, in teacher education, programming is connected to the school curriculum introduced in Kunnskapsløftet (LK20, 2019). To manage kids to learn to program, the concept of computational thinking was introduced, where algorithms are linked to systematic problem solving and support «thinking like a computer» (Wing, 2006). This approach is also understood as a way of experimenting and tinkering with technology (Csizmadia et al., 2015). Computational thinking is seen as a means to uncover a problem field and is necessary when specific sub-problems are to be solved. In teacher education and in schools, building and programming small robots have been introduced to facilitate different encounters between computational thinking and the affordances of a given technology. In such projects, there is a close connection between the robots' physical properties and the behavior implemented virtually in code. When students build and code their own robots, they open up for discussions about how automatons become influential on an individual level and affect us at the societal level. This approach may not be as relevant in teaching bio technicians or automation engineers programming. Still, in teacher education, it is very relevant to address the pupils' way of computational thinking and learning (Fojcik et al., 2020).

3.1.4. The choice of language

Another challenge in teaching programming is the language specifications. There are different programming languages, but all of them have English as a foundation. This might be difficult for non-English speakers. On the one hand, if some names and variables were called in the native language, it might help the students understand the program structure. On the other hand, if the students keep the entire code in English, there would be easier to seek help or search for information in the literature. An experiment at the University of Oslo was performed in 2000 (EasyIO, 2020), where the Java environment was translated into Norwegian. This product allows using most of the coding in Norwegian. Unfortunately, after every update to java-software (1-3 times a year), there is a need to update EasyIO.

Since the updates were not coordinated, it took a long time. During that time, the product could not be used. The delay and the continuous working amount were the reasons for discontinuing EasyIO. Another issue was that the students who used EasyIO during their studies had to learn the "common" English way after the studies were finished and started working. A similar situation happens to C#. Many additional programs show class structures graphically. However, the program descriptions are only in English. Using Norwegian names makes it more difficult than easier because the students need to learn Norwegian and English terms.

3.2. Teacher

3.2.1. Student-centered approach

The other aspect is shifting the focus from the teacher to the student. Teachers tend to base their teaching on their own needs or assumptions about the student's needs. As the Didactic Triangle represents, there is a connection between teachers and students and teachers and content (Berglund & Lister, 2010). It is not enough to just look at the content to teach. There needs to be a pedagogical approach to consider students' side of the process of learning.

3.2.2. Experiences with programming

Teaching programming means: "*teaching how to interact with a computing machine and how to deal with all the surprises that a machine can throw at you"* (Kak, 2009, p. 2). Therefore, teachers need to demonstrate interesting cases, show good practices, and instill the joy of programming in students. This can be done by someone who truly knows what he/she is doing, both in the subject's content and in pedagogical approaches. If, for example, a teacher with automation experience is teaching programming for biotechnology students, they would point out their own experience. Still, the results might not be comparable with the students. The same goes for professors that have not been involved in software development. Then, the only experience shared with students can be a theoretical one (Berglund & Lister, 2010).

3.2.3. Updating qualifications

"One of the greatest dangers in teaching is the routine, and habitual repetition of actions often observed in one's teachers or colleagues" (Czerepaniak-Walczak, 2014, p. 10). This is particularly evident in teaching engineering subjects related to computer science. This area is quickly changing time. Therefore, updating the teacher's qualifications and experience is an absolute must to avoid repetitions. Terms and concepts in computer science 50 years ago might not be used anymore. At the same time, the industry introduces many new terms and ideas that are more relevant for students when they apply for jobs after they finish studying.

3.2.4. Programming as a supplementary tool

In some subjects, programming is not the primary objective. Still, it is used as a tool that makes it possible to achieve other educational goals. Initially, this will emphasize what programming is used for and fewer concerns about efficiency, quality of code, etc. Following such an approach, one can later work more in detail regarding the actual code. Findings from Danish primary schools show that this approach may be fruitful. The schools have invested quite a lot in small robots, where the idea is that the robots can be used to teach pupils mathematics and English. Unfortunately, school practices show that robots do not work well in the intended topics. However, they are very well suited for introducing pupils to code (Bruun & Hasse, 2016).

Another example is when code is used as an artistic tool. When teaching mainly aesthetic topics, the teachers do not emphasize the logic and code quality as such. It is the outcome, what the code produces, that is considered most significant. When code is used to create a visual or auditory object that is not presented live, the program's efficiency is close to irrelevant as long as it produces a result. In other instances, time can be crucial, for example, when live coding music. One such example is Sonic Pi (https://sonic-pi.net), a code-based tool used to create music and live performances.

3.3. Student

3.3.1. Pedagogical approach

As mentioned in section 3.1, some students misunderstand how programming works. The first meeting with programming can be confusing. Many students start their learning process with memorization and creating habits they do not fully understand. For example, some students claim that: "*I do not know what this code sequence does, but without it, the program does not work*" or "*In all previous programming tasks, we have done this, so we continue to do it now as well*". This can be explained as reward and punishment in behaviorism. When the students choose a code sequence that works, they might use it again in a different setting to see if it is still working. This might create a "reward" for this habit, even if the student does not understand why it works (or does not work) (Ertmer & Newby, 2013). Presents that constant patterns, repetitive actions (not necessary with understanding), and signals for positive and negative responses can be desirable and helpful at the beginning of the course. Students have to learn names and definitions. But the programmer cannot rely only on repetition and memorization. It is necessary to have understanding, and problem-solving skills, to talk with others – to divide the problem into smaller parts, which are possible to (parallel) in a group.

3.3.2. Students' expectations

Most students want to acquire valuable knowledge for work after they finish their education. Still, they don't always know what is needed to achieve their goal; therefore, they rely on teachers/students and the descriptions of the subjects taught there. Students observe the world, use modern tools (mobile phones, PC, smart homes/watches/...), and often communicate that they would like to have something similar in their studies. For example, they say: "*Why should you learn two years of text programming (instead of graphical GUI) when NO ONE (outside the university) uses text programming*." Sometimes the students have interesting ideas that should be implemented in the course. In contrast, they at other times want to learn about big ideas like the Internet of Things and do not see that it is a more complex term than just the name.

There is a potential for an increased mismatch between the most common ways of programming, applied to solve more or less common problems and the programming of more complex systems. A potential future of coding envisions code and software written by or assisted by Artificial Intelligence and neural nets – Software 2.0 (Karpathy, 2017). Such systems will be able to translate the natural language to code across a number of programming languages.

A future where computer code is created through verbal dialogue with an AI can open programming to new fields and other practices. At the same time, this (see OpenAI and their demonstration of GTP3 - https://youtu.be/SGUCcjHTmGY). This continues an ongoing shift from lower to higher levels of abstraction over the years. With the development of AI and language models, there is a sharp increase in this development. This will still require a technical understanding and the ability to see the need and impacts of what one is building

and how a solution can become part of a whole. This can be greatly beneficial for innovation and progress in many areas, especially in fields where computers have had relatively little impact. Some artistic use may serve as examples.

A potential downside is that students that are not very interested in learning to code the traditional way will be able to skip this. It is also a fundamental question regarding how the models are trained. The skills needed to come up with a new, innovative solution will perhaps change, but hardly be less hard.

3.3.3. Teaching through Pandemic

An internal survey of Western Norway University of Applied Sciences showed that digital lectures make programming more difficult. Grades from last year's introductory programming exams show that many more students failed exams than usual before the previous year. There were the same teachers (and additional assistants) and the same time for teaching, practice, and labs. However, the form was different – the students could not meet on campus, and there was no physical contact with the teacher or the assistant. It looks like it was the only change that affected students significantly (table 1).

Table 1.
Comparison of student results in a programming course.

HVL programming (DAT 100)	2019 - blended	2020 digital online
Students accepted to the exam	92%	89%
Accepted students who passed the exam	84%	65%
Students who manage to score course	78%	58%

4. DISCUSSION

This chapter presents different points of view and different criteria for the content of programming, teachers that teach programming, and students that are learning to program. Today, teaching is not happening individually. There are still possible to have a relationship as a mentor-apprentice. Still, today's structure often involves many elements: teachers, students, administration, management, library, and IT services. Teachers should share experiences and help and motivate each other. Different people with their ideas, backgrounds, and point of view in good cooperation can do much more than each individual. To increase such synergies, this collaboration has to be organized. One approach is to develop and implement a system/rules that facilitate knowledge sharing and collaboration. If all want to do the same – it is not inspiring, and shared resources may distribute the work and allow individuals to contribute where they are strongest. Open-source projects with a shared codebase and distributions of tasks may serve as a potential model.

Teaching programming has many possibilities and requirements. The teacher's knowledge, the avoidance of problematic elements in teaching, and the use of modern and accessible tools require particular knowledge and skills. This does not seem to be necessary in every case for teaching programming. It seems advisable to divide programming courses according to the expected knowledge of the students. Programming students should have more theoretical and practical knowledge and skills. Students of other majors don't need the same theory, and basic knowledge of algorithmics, problem-solving, and commitment would suffice.

With sophisticated language models, like OpenAI Codex (OpenAI, 2022), this seems quite likely in the near future: Once a designer knows what to build, the act of writing code can be thought of as (1) breaking a problem down into simpler problems, and (2) mapping

those simple problems to existing code (libraries, APIs, or functions) that already exist. Most programmers probably find the latter tasks the least interesting part of programming. At the same time, this is where assistance by AI excels the most. In this scenario, coding becomes available as a service, and the most crucial skills will be defining and breaking down the problems that the programs are to solve. Important skills will then be training and weighting of the models, and the needed skills may be very different from what is commonly understood as programming today.

Table 2.
Summary of the challenges according to the Didactical Triangle.

	Content	**Teacher**	**Student**
Challenges	Anthropomorphizing metaphor	Student-centered approach	Pedagogical approach
	Different approaches	Experiences with programming	Students' expectations
	Specialized content for the course	Updating qualifications	Teaching through pandemic
	The choice of language	Programming as a supplementary tool	

5. CONCLUSIONS

What should be the content in programming courses? What is necessary knowledge, and for whom? The research question discussed in this study is: What can be challenging in introducing programming based on the Didactical Triangle: Content, Teacher, and Student? Real-world examples show that there is no "golden solution" to teaching programming that will fit every course, every teacher, and every student, but by addressing the didactical challenges, see table 2, there is a potential to teach computer programming in a way that will let students practice their skills and competencies.

In teaching computer programming, there are different expectations in scope (speed, standards, knowledge of technologies, libraries, programming environments) as well as the area (industry, banking, marketing, education), which shows that there are very different expectations, and it is impossible to meet them all in all courses. Some courses can be more effective for particular students, while others might reach other students. There is no "one size fits all", and different approaches to programming fit better to different students. The teachers should be aware of the didactical challenges and obstacles on various levels of teaching computer programming and choose relevant methods.

It is necessary to choose the right topic depending on what you want to teach the students. Many times, courses have very general names, which can mislead candidates. E.g., "advanced programming", "introductory programming", and "basics of programming". - what do they include? Object-oriented programming (another popular term) can be treated in several ways. Depending on the teacher, students can learn the whole subject or just a small part. Unfortunately, there is no standard naming for programming topics and what knowledge and skills they are developing.

A division into different types of programming courses - for computer scientists and for "others" - is suggested. Teachers require a different approach than computer scientists. Their purpose for education and the challenges they meet in their professional lives differ.

A programming course is a challenge for many students. It should be taught by a competent teacher with knowledge of the subject and pedagogy. It's easy to alienate students.

It's harder to motivate them if there are problems. In programming, almost every element builds on the previous ones. Lack of mastery of the primary material (for various reasons) will give rise to deficiencies in subsequent elements, and as a result - the student will "drop out". A solution might be to support the students with more follow-ups by the teachers. It shouldn't be that a teacher has a student for 1 semester, and then whatever happens.

Conversely, the student learns for a few years and then goes to work. Teachers should adapt their teaching to the student's needs, not do what suits them better. The use of the (national) language should be considered - whether it will bring more benefits or problems in the following years. We probably also have to consider the impact of language models on creating code. This can lead to fundamental changes where the definition of problems, their structure, and the prioritizing among different solutions will be of much greater importance than the code itself.

REFERENCES

Berglund, A., & Lister, R. (2010). Introductory programming and the didactic triangle. In T. Clear & J. Hamer (Eds.), *Proceedings of the twelfth Australasian conference on computing education – Volume 103* (ACE'10) (pp.35–44). Australian Computer Society. http://hdl.handle.net/10453/16619

Bettin, B., Jarvie-Eggart, M., Steelman, K. S., & Wallace, C. (2022). Preparing first-year engineering students to think about code: A guided inquiry approach. *IEEE Transactions on Education*, *65*(3), 309–319. https://doi.org/10.1109/TE.2021.3140051

Biggs, J. (2012). What the student does: Teaching for enhanced learning. *Higher Education Research & Development*, *31*(1), 39–55. https://doi.org/10.1080/07294360.2012.642839

Blackwell, A. F. (2002). What is programming? In J. Kuljis, L. Baldwin & R. Scoble (Eds), *Proceedings of the 14th workshop of the Psychology of Programming Interest Group* (PPIG), 204–218. Retrieved from https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fffaed8f98da11d4c6fb3d692dc0ffd41736f209

Brown, N. C., & Wilson, G. (2018). Ten quick tips for teaching programming. *PLoS computational biology*, *14*(4), e1006023. https://doi.org/10.1371/journal.pcbi.1006023

Bruun, M. H. & Hasse, C. (2016). Studying robots in the wild. In J. Seibt, M. Nørskov & S. Andersen (Eds.), *What social robots can and should do: Proceedings of Robophilosophy 2016 / TRANSOR 2016*, (vol. 290), 373–377. IOS Press, Incorporated.

Cawthorne, L. (2021). Invited viewpoint: teaching programming to students in physical sciences and engineering. *Journal of Materials Science*, *56*(29), 16183–16194. https://doi.org/10.1007/s10853-021-06368-1

Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., & Woollard, J. (2015). *Computational thinking - A guide for teachers*. Retrieved from https://eprints.soton.ac.uk/424545/1/150818_Computational_Thinking_1_.pdf

Czerepaniak-Walczak M., (2014), Miedzy teorią a praktyką, Funkcje koncepcji pedagogicznych w pracy nauczycieli i nauczycielek. [Between theory and practice. Pedagogical concept functions in teachers work]. *Refleksje 6*, 10–14.

Demir, F. (2022). The effect of different usage of the educational programming language in programming education on the programming anxiety and achievement. *Education and Information Technologies*, *27*(3), 4171–4194. https://doi.org/10.1007/s10639-021-10750-6

De Raadt, M., Watson, R., & Toleman, M. (2009). Teaching and assessing programming strategies explicitly. In M. Hamilton & T. Clear (Eds.), *Proceedings of the eleventh Australasian conference on computing education – Volume 95 (ACE 2009)*, 45–54. Australian Computer Society. Retrieved from https://research.usq.edu.au/download/c4fbd62549f5c273e706c0384353bc7878d4564b76f530eb34451529e8b5b5a5/1061813/DeRaadt_Watson_Toleman_ACE2009_PV.pdf

Dijkstra, E.W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM, 32*(12), 1398–1404. Retrieved from https://www.psy.gla.ac.uk/~steve/educ/dijk/EWD1036.v4.pdf

EasyIO, (2020). *Instalasjon av programpakken easyIO [Instalation of programmingpackage easyIO*]. https://nettressurser.no/java/Java-og-easyIO

Ertmer, P. A., & Newby, T. J. (2013). Behaviorism, cognitivism, constructivism: Comparing critical features from an instructional design perspective. *Performance improvement quarterly*, *26*(2), 43–71. https://doi.org/10.1002/piq.21143

Fojcik, M. K., Fojcik, M., Sande, O., Refvik, K. A., Frantsen, T., & Bye, H. (2020). A content analysis of SOLO-levels in different computer programming courses in higher education. In *Norsk IKT-konferanse for forskning og utdanning* (No. 4). Retrieved from https://ojs.bibsys.no/index.php/NIK/article/view/823

Fojcik, M., Galek, J., & Fojcik, M. K. (2017). IKT kompetanse blant studenter. Er vi klar for framtiden? [ICT-competencies in students. Are we ready for the future?]. In R. Lyng & M. M. Jakobsen (Eds.) *Proceedings for the MNT-Conference*, 204–207.

Ginat, D., & Menashe, E. (2015). SOLO Taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 452–457). https://doi.org/10.1145/2676723.2677311

Griffin, P., & Care, E. (2015). The ATC21S method. In P. Griffin & E. Care (Eds.), *Assessment and teaching of 21st century skills. Educational assessment in an information age*, 3–33. Springer. https://doi.org/10.1007/978-94-017-9395-7_1

Hartree, D. R. (1950). Automatic Calculating Machines. *The Mathematical Gazette, 34*(310), 241–252.

Kak, A. (2009). *Teaching programming*. Retrieved from https://engineering.purdue.edu/kak/TeachingProgramming.pdf

Kansanen, P., & Meri, M. (1999). The didactic relation in the teaching-studying-learning process. In B. Hudson, F. Buchberger, P. Kansanen, & H. Seel, (Eds.), *Didaktik/Fachdidaktik as Science (-s) of the Teaching profession (TNTEE)*, *2*(1), 107–116.

Karpathy, A. (2017). *Software 2.0*. https://karpathy.medium.com/software-2-0-a64152b37c35

LK20 (2019). *Kunnskapsløftet 2020, Curricula in English*. Utdanningsdirektoratet. https://www.udir.no/in-english/curricula-in-english/

Malik, S. I., Tawafak, R. M., & Shakir, M. (2021). Aligning and assessing teaching approaches with solo taxonomy in a computer programming course. *International Journal of Information and Communication Technology Education (IJICTE)*, *17*(4), 1–15. https://doi.org/10.4018/IJICTE.20211001.oa5

Martin, A. (2006). A European framework for digital literacy. *Nordic Journal of Digital Literacy, 1*(2), 151–161. https://doi.org/10.18261/ISSN1891-943X-2006-02-06

merriam-webster dictionary (2020). https://www.merriam-webster.com/

OpenAi (2022). OpenAI Codex. https://openai.com/blog/openai-codex/

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Selby, C. C. (2014). How can the teaching of programming be used to enhance computational thinking skills? (Doctoral dissertation, University of Southampton, Southampton, United Kingdom). Retrieved from https://eprints.soton.ac.uk/366256/

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. L. (2008). Going SOLO to assess novice programmers. In *Proceedings of the 13th annual conference on innovation and technology in computer science education* (pp. 209–213). https://doi.org/10.1145/1384271.1384328

Sheard, J., & Carbone, A. (2007). ICT teaching and learning in a new educational paradigm: Lecturers' perceptions versus students' experiences. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88* (pp. 109–117). Retrieved from https://www.researchgate.net/profile/Judy-Sheard-2/publication/228394103_Ict_teaching_and_learning_in_a_new_educational_paradigm_lecturers'_perceptions_versus_students'_experiences/links/563c804a08ae405111a9e048/Ict-teaching-and-learning-in-a-new-educational-paradigm-lecturers-perceptions-versus-students-experiences.pdf

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? In *Proceedings of the second international workshop on Computing education research* (ICER'06), 17–28. https://doi.org/10.1145/1151588.1151593

Vuorikari, R., Punie, Y., Carretero, S. & Van den Brande, L. (2016). *DigComp 2.0: The Digital Competence Framework for Citizens. Update Phase 1: the Conceptual Reference Model.* Retrieved from https://publications.jrc.ec.europa.eu/repository/handle/JRC101254

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM Sigcse Bulletin*, *28*(3), 17–22. https://doi.org/10.1145/234867.234872

AUTHOR(S) INFORMATION

Full name: Marcin Fojcik
Institutional affiliation: Western Norway University of Applied Sciences
Institutional address: HVL, campus Førde, Svanehaugvegen 1, 6812 Førde
Short biographical sketch: Professor Marcin Fojcik's research interests are communications and automation. This includes basic and applied research through many projects with Quality of Service, Service Oriented Architecture, and different types of industrial communication: from real-time protocols and Factory 4.0 to the Internet of Things. He also works with pedagogical and didactical methods in teaching Science, Technology, Engineering, and Mathematics (STEM) subjects.

Full name: Martyna Katarzyna Fojcik
Institutional affiliation: Volda University College
Institutional address: Joplassvegen 11, 6103 Volda
Short biographical sketch: Martyna Fojcik is a Ph.D. student at the University of Agder. In her project, she is researching in-service mathematics teachers' conceptualization of programming in mathematics education. She has an M.Sc. degree in Mathematics Education from the University of Agder. Since fall 2019, she is working at Volda University College teaching mathematics in teacher education programs and in courses for postgraduate in-service teachers in mathematics and programming. She is interested in inquiry-based learning, argumentation in mathematics, programming for mathematics education, digital tools in teaching and learning.

Full name: Sven-Olai Høyland
Institutional affiliation: Western Norway University of Applied Sciences
Institutional address: HVL, campus Bergen, Inndalsveien 28, 5063 Bergen
Short biographical sketch: Associate Professor Sven-Olai Høyland has a PhD degree in Computer Science and is mainly teaching different courses in programming and algorithms.

Full name: Jon Hoem
Institutional affiliation: Western Norway University of Applied Sciences
Institutional address: HVL, campus Bergen, Inndalsveien 28, 5063 Bergen
Short biographical sketch: Jon Hoem is associate professor at the Department of Arts Education at Western Norway University of Applied Sciences. He teaches and conducts research in new, digital media with an emphasis on the relationship between digital media and physical environments. His latest book is on Digital media and materiality. He also does projects in schools, involving teachers and pupils in STEAM-education.