

ENHANCING THE PROFESSIONAL COMPETENCE OF FUTURE COMPUTER SCIENCE TEACHERS THROUGH PROGRAMMING LANGUAGES

Rakhmatov Mirzo Mukimovich

Independent Researcher at Bukhara State Pedagogical Institute

Abstract. This article examines the theoretical and methodological foundations for enhancing the professional competence of future computer science teachers through the effective teaching of programming languages. It highlights the role of programming education in developing algorithmic and computational thinking, problem-solving abilities, practical coding skills, creativity, and digital literacy. Particular attention is given to professionally oriented programming tasks, collaborative projects, digital learning platforms, code-testing tools, and reflective assessment. The proposed methodological approach integrates programming knowledge with pedagogical and methodological skills, enabling future teachers to design algorithms, develop software products, explain programming concepts clearly, identify and correct coding errors, and organise effective programming lessons.

Keywords: future computer science teachers, professional competence, programming languages, programming education, algorithmic thinking, computational thinking, digital competence, competency-based approach, project-based learning, problem-solving skills, teaching methodology.

Introduction. The increasing integration of software systems, artificial intelligence, data processing, and automated technologies into different areas of society has changed the professional requirements imposed on computer science teachers. Modern teachers are expected to prepare learners not only to use digital devices but also to understand computational processes and create digital products. Accordingly, programming languages have become an important instrument for developing the professional competence of future computer science teachers. Programming languages provide an environment in which theoretical concepts are transformed into observable and testable results. When writing a program, a student must analyse the problem, determine the input and output data, identify logical relationships, construct an algorithm, select appropriate language structures, and verify the result. This sequence develops disciplined thinking and teaches students to make decisions on the basis of evidence. Therefore, programming education should be considered an important means of professional preparation rather than merely a technical part of the computer science curriculum.

The professional activity of a computer science teacher requires a combination of subject knowledge and pedagogical mastery. A specialist may be able to develop a complex program but may experience difficulties in explaining its logic to learners. Conversely, a student with good communication skills may be unable to organise programming instruction without sufficient knowledge of algorithms and software development. Higher pedagogical education must therefore establish a meaningful relationship between programming proficiency and the ability to teach programming. Programming languages differ in their syntax, structure, application areas, and level of complexity. For this reason, teaching should not be limited to memorising the commands of one particular language. Future teachers need to understand universal programming concepts that can be transferred from one language to another. Variables, data types, branching, repetition, functions, data structures, modularity, and object-oriented principles retain their conceptual significance even when their syntactic forms differ. Mastering these concepts enables future teachers to adapt to technological changes and independently learn new programming languages. An important problem in teacher preparation is the gap between university programming courses and classroom practice. Students frequently solve standard mathematical problems but have limited experience in creating teaching materials, analysing learners' code, explaining common misconceptions, or planning programming lessons. As a result, their technical knowledge may not automatically become professional teaching competence. To address this issue, programming courses should include assignments that model the actual responsibilities of a computer science teacher. The enhancement of professional competence through programming languages consequently requires the revision of educational content, classroom activities, practical assignments, assessment procedures, and teaching practice. The learning process should prepare future teachers to select appropriate programming environments, organise explanations, manage learners' cognitive difficulties, develop educational applications, and evaluate programming outcomes. This approach makes it possible to connect technological preparation with the realities of professional pedagogical activity.

Programming languages perform several functions in the professional development of future computer science teachers. First, they serve as a means of formalising information. A programming language requires every instruction and condition to be expressed precisely. Ambiguous formulations, incomplete

sequences, and logical contradictions usually lead to incorrect results. This characteristic develops accuracy, consistency, and attention to detail.

Second, programming languages provide opportunities for modelling objects and processes. Students can create models of mathematical calculations, physical phenomena, economic indicators, and educational situations. For example, a program may calculate students' assessment results, generate individual tasks, visualise statistical information, or simulate the execution of an algorithm. Such activities demonstrate the interdisciplinary applications of programming and prepare future teachers to establish connections between computer science and other subjects. Third, programming promotes the formation of diagnostic abilities. The process of debugging requires the programmer to identify the source of an error, examine intermediate results, formulate assumptions, and test possible corrections. These actions are closely related to a teacher's professional responsibility to diagnose learners' difficulties. A future teacher who can systematically analyse programming errors is better prepared to understand why a learner has produced an incorrect solution and what kind of methodological assistance is required.

The professional competence of future computer science teachers can be improved when every programming topic is connected with its teaching methodology. For instance, after studying conditional operators, students should not only write programs containing conditions but also determine how to explain branching to school learners. They may compare verbal descriptions, flowcharts, pseudocode, and program code and then select the clearest form of presentation for a particular age group. When studying loops, future teachers should examine common learner errors, including incorrect initial values, unsuitable conditions, infinite repetition, and errors in updating control variables. They should prepare examples that demonstrate these problems and develop questions that help learners independently identify the cause of an incorrect result. This approach transforms error analysis into a professional teaching skill. The topic of functions can be connected with the principles of decomposition and modular teaching. Students learn to divide a complex problem into smaller subproblems and implement each part as a separate function. From a pedagogical perspective, they also learn to divide complex educational content into logically complete learning units. Thus, the structure of programming activity can contribute to the development of lesson-planning abilities.

Data structures provide opportunities for teaching students to select an effective form of information representation. Future teachers should understand that the choice between a list, dictionary, set, table, or class depends on the characteristics of the problem. They should also be able to design educational examples that reveal these differences. Such examples may involve student records, subject grades, electronic library data, test questions, or attendance information.

The content of programming education should be organised according to conceptual complexity and professional significance. Instruction may begin with algorithmic structures and simple program construction. At this stage, students learn to define a problem, identify data, formulate a sequence of actions, and predict the program's result before executing it. The next stage should include functions, collections, file processing, and modular programming. These topics enable students to develop larger and better-organised programs. File-processing tasks may use educational data such as attendance records, test results, or student portfolios. Such contextualisation helps learners understand the professional value of the acquired programming skills. Object-oriented programming may be introduced after students have developed a stable understanding of procedures and data structures. Classes and objects can be explained through familiar educational entities such as a student, teacher, subject, lesson, assignment, or assessment record. For example, a "Student" class may contain personal data, grades, and methods for calculating an average score. This type of modelling makes abstract programming concepts more accessible. The final stage should involve the integration of programming concepts into complete software products. Students may develop applications for organising quizzes, presenting teaching materials, visualising algorithms, recording assessment results, or supporting extracurricular computer science activities. Such work demonstrates whether they can combine separate programming concepts into a purposeful and functional system.

Microteaching is an effective way to connect programming with pedagogical practice. A student prepares and conducts a short lesson on a selected programming concept, while other group members act as learners. The lesson may focus on explaining variables, conditions, loops, functions, arrays, or classes. After the activity, participants analyse the clarity of the explanation, the appropriateness of examples, classroom interaction, and the accuracy of technical content. Code-analysis activities also have considerable professional value. Students are given examples of correct, inefficient, or faulty programs and asked to explain their behaviour. They identify syntactic, logical, and semantic errors and then

propose corrections. More importantly, they determine how the error could be explained to a school learner without simply providing a ready-made answer.

Another activity involves designing a system of differentiated exercises. Future teachers prepare tasks for learners at basic, intermediate, and advanced levels. Basic exercises may involve completing a code fragment or predicting output. Intermediate tasks may require modifying an existing solution. Advanced assignments may involve designing an original algorithm or comparing several solutions. This process teaches future teachers to consider differences in learners' abilities. Students may also create methodical commentaries for program code. In addition to technical comments, they prepare explanations of the learning objective, required prior knowledge, anticipated difficulties, guiding questions, and assessment indicators. In this way, a program becomes part of a complete educational resource rather than an isolated technical product.

Several conditions are necessary for the successful use of programming languages in professional preparation. First, programming and teaching-methodology courses should be coordinated. Their content, assignments, and expected outcomes must complement one another. A programming topic studied in one course should later be examined from a methodological perspective. Second, practical sessions should provide sufficient time for independent work. Programming competence cannot be developed solely by observing a teacher's demonstrations. Students need opportunities to make decisions, encounter difficulties, test ideas, and revise their solutions. The teacher's role is to organise appropriate support without removing the intellectual challenge of the task.

Third, feedback should be timely and specific. General statements such as "incorrect program" do not explain what must be improved. Effective feedback should indicate whether the problem concerns algorithm design, language syntax, data selection, testing, code organisation, or explanation. Students should then be allowed to revise their work. Fourth, academic integrity must be maintained. The availability of online solutions and code-generation systems creates a risk of mechanical copying. Oral code explanations, individual modifications, development histories, practical demonstrations, and reflective reports can be used to confirm students' understanding. The purpose is not simply to control students but to ensure that digital assistance contributes to genuine learning. Finally, the university should provide a stable technological environment, including suitable computers, programming software, educational platforms, and access to electronic resources. However, infrastructure alone does not guarantee effectiveness. The central requirement is a carefully designed system in which every technological tool supports a defined professional learning outcome. Thus, programming languages can contribute significantly to the professional competence of future computer science teachers when technical learning is systematically connected with pedagogical activity. The effectiveness of this process depends on professionally oriented content, progressive task complexity, purposeful error analysis, teaching practice, comprehensive assessment, and reflective improvement.

References

1. Mamarajabov, M. E., Toshtemirov, D. E., & Yuldashev, O'. A. (2023). *Informatika o'qitish metodikasi: darslik*. Toshkent: Bookmany Print.
2. Yuldashev, U. Y., Boqiyev, R. R., & Zokirova, F. M. (2005). *Informatika o'qitish metodikasi: o'rta maxsus, kasb-hunar ta'limi muassasalari uchun qo'llanma*. Toshkent: Talqin.
3. Tolipov, O'. Q., Usmonboyeva, M. (2006). *Pedagogik texnologiyalarning tatbiqiy asoslari*. Toshkent: Fan.
4. Grover, S., & Pea, R. (2013). Computational thinking in K-12: A review of the state of the field. *Educational Researcher*, 42(1), 38-43.