International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 7 (2015) pp. 16491-16504
© Research India Publications
http://www.ripublication.com

Role of The Algorithm In Introductory Programming Courses

Teodosi K. Teodosiev[1] and Anatoli M. Nachev[2]
[1]Department of Mathematics and Computer science, Shumen University "Bishop K. Preslavski", Universitetska 115, Shumen, Bulgaria, e-mail: t.teodosiev@fmi.shu-bg.net
[2]Cairnes Business School, National University of Ireland, Galway, Ireland, e-mail: anatoli.nachev@nuigalway.ie

Abstract

This study explores introductory programming courses. We discuss the role of careful algorithmization when presenting the teaching material. Due to the nature of the tasks, which is training, is difficult to convince novice programmers in the benefit of prior consideration of the approach to the implementation and that process of reasoning carried out in the development process of the algorithm serves as a logical basis of the final form of the program. Therefore, an important methodological issue is using a careful selection of tasks and examples, which stimulate students to approach to the problems considering and estimating different algorithm alternatives, before moving to coding. In order to illustrate our point, we provide teaching example analysing the pedagogical outcomes. Reference is made between the levels in Bloom's taxonomy and the student abilities in algorithmization and programming. Even separate levels of the Bloom's pyramid can be discussed in the context of algorithmization and algorithm implementation, which in fact is the programming itself. From the comments made here, it becomes clear that the main focus of the introductory programming course should be directed towards the algorithmization, which requires logical and mathematical culture.

Keywords: introductory programming courses, algorithmization, Bloom's taxonomy, style of programming

Introduction

Often we observe that students who are beginners in programming start entering code straight after having the assignment, attempting and hoping to find a correct solution immediately after that. They skip the algorithm analysis stage, which is important to implement a 'good' algorithm (similar problems, but in other subjects have been discussed in [1, 2]). Students aim to put together just a working program, no matter it is readable enough or whether it may become more efficient. These observations is also valid with more advanced students taking place in programming contests - even they rarely seek to ease their work by consideration of building an appropriate algorithm first. Due to the nature of the tasks, which is training, is difficult to convince novice programmers in the benefit of prior consideration of the approach to the implementation and that process of reasoning carried out in the development process of the algorithm serves as a logical basis of the final form of the program.

The same phenomenon can be observed in math teaching. Years ago, the need of intensive and heavy computations of algebraic expressions led to striving for simplification of the expressions in order to minimize and optimize the calculations. Today, this is not practiced often, because the dealing with complexity is not issue for the modern computers and IT technologies. So, a point might be: once we have computers, which are powerful enough, why should we care about the algorithm? The new technologies, along with their many advantages have a significant disadvantage: users' behaviour becomes more consumption-driven, makes people lazier, and does not encourage creative thinking and aspirations. Common IT activities tend to be more focused to social networks and the phenomena and processes, which surround us, including teaching materials subject to study, are taken at the consumer level, which does not make the students to be active and thinking. But this is what the education seeks, particularly training in programming.

The set of tasks and examples used in an 'Introduction to Programming' course are simple enough due the requirement to be introductory. In that sense they can't force students to approach to the problem building effective solutions. Therefore, an important methodological issue is using a careful selection of tasks and examples, which stimulate students to approach to the problems considering and estimating different algorithm alternatives, before moving to coding.

The paper is organized as follows: Section 2 discusses the algorithmic culture. Section 3 is dedicated to the specifics of teaching in programming. Section 4 discusses the role of a well considered algorithm by an indicative example. Section 5 provides a link between cognitive levels in the Bloom's taxonomy and levels of implementation of the algorithm into a program. Section 6 provides conclusions.

Algorithmic Culture

Problem solving is a mandatory element of training in programming. In the process of finding solutions, the students master the skills and habits for applying the theoretical knowledge into practice. Moreover, the ability to solve problems is an indication that the theoretical knowledge is well understood. In conditions of the so named 'active approach' [3, 4, 5] to training, the core and the essence of the educational activity is solving tasks [6]. The term 'learning task' should be interpreted in a broader meaning - as any other - the solution is aims achieving specific goals in education. In practical terms, the task is explicit or implicit question, the answer of which is not obvious and should be found in stages.

In the context of the training process, a solution of a task is not really the goal; it is just a mean to reach a broader objective of formation of action modes as part of the programming skills. These models can be built by providing guidance and estimates from the tutor, they could be a result of unguided trial-and-error activities followed by estimation or self-estimation. The action models are the real value of the training process, not finding the solution itself. Actually, the solution is only an indicator that the action model has been developed properly.

The foundation of solid mathematical skills combined with preliminary analysis of the task is a key success factor and prerequisite for a correct action modelling process modeling process, as opposed to attempting the task by direct coding in an integrated development environment (IDE) of a programming language. Working on the program, the programmer, especially beginners, should respect the fact that the program they design is made not only for the user, but also help to the programmer himself [7].

Programmers must also be conscious, that in the context of the software development methodologies, the change management and maintenance stage of the software lifecycle, presume that other teams of programmers will review and possibly modify their code in future. This is particularly important and relevant to the modern trends of building reusable software units and libraries, which are usually involved in integration and adaptation in various software projects. A mandatory feature of the contemporary software, therefore, is readability, integrity, well defined interfaces, input and outputs, eliminated side effects, rigorous encapsulation in the context of the object-oriented (OO) programming paradigms, and efficient usage of the computing resources. All these features require much thinking, much planning, careful decision making, adoption of the best practices accumulated within software companies, expert teams, and the community as a whole. At the bottom line, the introductory course in programming should convince the students, that the bricks and mortar of those skills are put together in a solid foundation right here and right now.

In terms of readability, the programming is a process much closer to the composition of a natural language text, than it might seem at first glance. Of course, this is not about abstract literary texts, these are more informative texts. When working on them, attracting the attention of readers just needed the quality of the show. The natural languages can be used with variable accuracy towards expressing thoughts - whether very successfully, or not too much - but precise expression of thought is absolutely necessary when creating an algorithm or composing a program.

This is the reason to consider who the readers of the program are. The most obvious reader, which springs in mind first, is the machine itself. From machine's point of view, efficiency is the most important quality criteria of the program text. Addressing this quality requirement in the programming courses makes clear that deeper we get into the subject of programming, the bigger this problem becomes. E. Dijkstra [8] comments quality of computer programs: ‘on the one hand... programs could have a compelling and deep logical beauty, on the other hand … forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation’.

There are more readers of the program, some of which will review it in future as part of the change management (incl. the author himself), but in some circumstances, a second reader may need to review the program at the moment of composition. An interesting example is related to the modern agile software development methodologies [9], particularly Extreme Programming (XP), where better efficiency is achieved by 'pair programming'. XP requires that two programmers work at one computer, where one composes the code, the other watches over and controls the first one; they can switch their roles from time to time. The motivation for that approach is 'two heads better than one'.

We should be conscious of the difference between the programs used for training and programs designed in real software development environment. The tutor composes programs for training; therefore they must be easy to understand.

Scientists in education methodology from 1970s paid particular attention to the influence of computers and programming on the educational content. They pointed out that the algorithmization concept lies in the basis of programming, defined as a process of developing and describing an algorithm by the means of a specific language. Many human activities can be formalized and described as a process, which follows a specific algorithm. There are, however, many areas where algorithmization is not applicable, such as activities related to creativity and subjective decision making (including the process of creating algorithms!). The notion of the learners what an algorithm means is formed implicitly during study of various disciplines, most important of which is math. However, with the advent of computers and Information and Communications Technologies (ICT), that notion is becoming more self-contained and influenced by the modern culture and the information age. Since mid 1980s, the primary task of programming courses has been formation of algorithmic culture of learners. Actually, the ability to define, put together, verify, and execute correctly a mathematical algorithm always been a major component of the mathematical culture of the student, although the term ’algorithm’ may not been mentioned. In this context, lack of functional literacy, which is ability to extract and interpret information from text, could make difficult building a correct algorithm and program.

In conclusion, we can summarize that the algorithmic culture is essential for skills formation in students necessary to perform algorithmization, which entails well structured and efficient computer programs.

Teaching Programming

Programming, considered as just act of coding of ready algorithms without paying attention to their meaning and potential for enhancement, cannot be the goal of a programming course, either general of specialised. The main objective of the teaching should not be focused to the syntactic and semantic aspects of the programming language, but to providing knowledge and skills for algorithmization. The ultimate purpose of the course is developing skills for solving classes of tasks and problems, given that this is doable by the means of the programming language. Abilities to compose programs are related mostly with having knowledge in math and logic, not mastering the syntax of the programming language, despite the language syntax and semantics are also related to the logic.

An important component of the algorithmic literacy is the ability to develop algorithm fragments in the best way they should be developed, not in the way which first springs in mind or being driven by sometimes vague task formulations and/or user expectations. The task formulation is not always precise. Sometimes, real-life software projects gather quite unclear or misleading user requirement, as the user is not always aware of what they want from the outset. The problems of which emerge on the surface at a later stage, when much time effort, and resources have been spent in vain. In summary, the teaching in programming should encourage the domination of the driven-by-purpose approach over the driven-by-expectations one, in situations where the two collide for the task in question.

A practical tool that might be found useful for building algorithmization skills is utilizing debugging techniques in integrated development environment (IDE), particularly the step-by-step tracing of the algorithm executions. This allows the developer to better understand the algorithm, detect flaws and logical errors, and ultimately, building experience in algorithmization.

Introductory programming courses aim to give students a basic knowledge in programming, but at the same time they place a ground for more advanced skills and knowledge. In these courses, paying attention to a good programming style is particularly important for the process of building solid programming skills.

The emphasis should be placed on building skills to solve problems, not learning the specifics of the programming language. The aim is to focus on the thinking process, shaping a thinking discipline, which helps avoid unnecessary complications in the training process. N. Wirth [10] says that programming presents as a discipline with its own qualities as a methodology for constructive considerations, applicable to any problem amenable to algorithmic solution.

Planning an introductory programming course should not strive to teach the programming language in its fullness. We need to bear in mind that the primary objective of the study of the course is the acquisition of knowledge and skills for algorithmization and learning methods and approaches to solutions of classes of problems. As Nicklaus Wirth states in [10] '... an academic institution's ultimate goal must be much wider than the mastery of a language. It must be nothing less than the art of designing artefacts to solve intricate problems. Some call it the art of constructive thinking.'

Students are expected to understand and know the basic concepts of programming, but they also have to understand the algorithmic approach to solving problems. By itself, this task is too complex [11, 12] and often leads beginners to misunderstandings and disillusioned. Learning programming requires from students certain mental effort, concentration, attention, logic, and imagination. The primary requirement, however, is having algorithmic thinking. In a programming course, the students should learn clearly and accurately how to implement the algorithm of their actions, to be able to record it correctly on paper and then enter the code to the computer. This would gradually discourage students to take wrong moves, inaccurate or improper decisions while creating an algorithm.

Basic concepts in programming, particularly control structures (such as sequence, selection, and iteration), mechanisms for aggregation (array, structure, union), pointers, functions, etc. can be discussed in their abstraction in parallel with the studying the formal syntax of a programming language [13]. Learning a programming language should be in the context of the problem solved, therefore new stuff should be taught as soon as it is needed for that particular task. Ultimately, the goal of teaching is to make students to learn basic programming principles and concepts, regardless of the programming language taught.

The logical sequence of presenting the course material and level of difficulty of the topics suggest that teaching should start with a discussion about models first, followed by algorithms, and then basics of programming. Algorithmization as part of programming is a central element of the course. After building some basic skills of structural algorithmization, teaching should move on to the language specifics. According to Van Tassel [14], the solutions of many problems have roots in some mathematical models. The teaching should not spare time discussing the math model in depth. This would help for better understanding the problem and to find a natural approach to its solution. The algorithmization includes a careful determination of the algorithm as per the math model, determining performance requirements, and designing the data structures required for that solution.

Example

As noted in the introduction, due to the nature of the training examples of an introductory programming course, it is difficult to convince students to approach to a problem by a well-considered solution. Gal-Ezer et al. [15] describe an interesting approach to this educational problem - they introduce the concept of complexity of the algorithm in a 3-rd level college course 'Introduction to Programming'. Relatively early introduction of that concept encourages students to consider alternative algorithms, to analyze them and formulate them correctly. Of course, such an early introduction of complexity can cause problems - tasks in the first intro classes are very simple and easy and do not contribute to convincing students that the more efficient the algorithm is the better. Furthermore, analysis of efficiency requires certain knowledge in math, which students don't always have.

It is also the case that wrong skills and habits that students may have acquired are difficult to be changed post-factum. This justifies the importance of a careful selection of the training examples, which can avoid the aforementioned problem regardless of their low complexity. Let's consider an example, which illustrates recommendations for improving efficiency of a program, and finally convincingly shows the benefit of a well-considered algorithm.

Example: Create a program that inputs two integers a and b, and displays the average of all integers belonging to the interval [a, b].

Here we illustrate four versions of the program, each of which improves the previous one from point of view of efficiency, simplicity, and readability. Also, the first two versions allow discussion on loops optimization.

The authors' observations show that nearly 90% of the students approach to the problem straight, attempting to put code together and relying on their background of what average is.

By definition, the average of a list of numbers is the sum of the numbers divided by the size of the list, in other words the arithmetic mean. This definition contains an embedded algorithm, which is common and can be applied to other tasks related to finding average. For this task, one need to find the sum of the integers in the interval [a, b] and the number of integers in that interval. A straight solution is to accumulate iteratively the sum of integers and to count the integers, similarly. The implementation requires two loops and a parameter, which continuously passes through the numbers of the interval in order to find the sum and the size.

Finding the quotient of two numbers deserves a discussion about features of the operation division with integer operands in C ++ and also type conversion mechanism, both implicit and explicit (casting).

```
//Version 1 C++
void main()
{
int sum=0, count=0, i, a, b;
cin>>a>>b;
for (i=a; i<=b; i++)
sum+=i;
for (i=a; i<=b; i++)
count++;
double avg=(double) sum/br;
cout<<avg<<endl;
}
```

Loops are one of the most important factors, which affect the algorithm efficiency, complexity, and execution time. This is because the body statements iterate and execute many times and any reduction of calculations, even a minimal, multiplies the effect of reduction many times. Minimization of the calculation in the body of a loop should be an objective.

One way to reduce the number of loops to merge two or more cycles in one. Thus reduce execution time and memory needed. This is often possible if students carefully analyze the task before coding.

Considering the code above, it can be noticed that version 1 uses two loops with same parameters and header lines. Joining the bodies of those loops into one loop improves the effectiveness of the program and shortens the code, as illustrated below.

```
// Version 2
void main()
{
int sum=0, count=0, i, a, b;
cin>>a>>b;
for (i=a; i<=b; i++)
{
sum+=i;
count++;
}
double avg=(double) sum/br;
cout<<avg<<endl;
}
```

Further analysis of the algorithm above shows that the numbers in the interval [a, b] are consecutive, which allows calculation of the number by the formula (b-a+1)/2. The analysis also shows that the sum of the numbers is actually sum of a finite arithmetic progression. Making this conclusion is possible only if the students' math background is present and solid. If that knowledge is missing, the teacher's analysis may encourage students to get a hold of that fact by observation and logic. Indeed, the sums of each pair of numbers which are symmetrical to the middle of the list are same. Therefore, the sum of the numbers in the interval [a, b] is equal to the sum of a pair of numbers multiplied by the number of pairs. The approach is applicable to the calculation of other Gaussian sums. Version 3 below illustrates these considerations.

```
// Version 3
void main()
{
int sum, count, a, b;
cin>>a>>b;
sum=(a+b)*(b–a+1)/2;
count=(b–a+1);
double avg=(double)sum/count;
cout<<avg<<endl;
}
```

Improved efficiency in Version 3 is obtained by careful analysis and evaluation of the algorithm. An alternative approach to reaching those conclusions in class and obtaining the simple formula embedded in the algorithm is by detailed recording of the mean expression, followed by its simplification while paying attention to equal terms.

Same conclusion can be made if the discussion moves one step further to considering the fact that if the sum of each symmetrical pair is the same, then the average of each pair (sum divided by 2) is the same. This entails the next version of the program:

```
// Version 4
void main()
{
int a,b;
cin>>a>>b;
double avg= (double) (a + b)/2;
cout<<avg<<endl;
}
```

Apparently, each version of the solution improves the efficiency of the previous one, at the same time getting shorter. Teachers and students must pay attention to not only correctness of the programs, but their quality, as well. As Skupiene [16] says, in education, different qualities and styles of programs should be treated differently. A responsible attitude to the style of programming is very important for those who will participate in future programming contests, particularly in their first stages of study.

Let's consider one more example very briefly:

Example: Write a program that finds the number of even digits in the record of a given natural number.

Here are two possible solutions:

A. Check if the number is even (that is its last digit is even) and if yes, add 1 to the counter of even digits, and then repeat the same for the number without its last digit. This process continues until the number remaining hits 0.

```
//Version A in C++
void main()
{
int a, br = 0;
cin >> a;
while ( a )
{
if (!( a % 2 )) br++;
a /= 10;
}
cout << br << endl;
}
```

B. Extracting and storing the digits of the number in an array and counting the even elements of that array.

```
// Version B
void main()
{
int a, i = 0, ch[10] = {0}, br = 0;
cin >> a;
while ( a )
{
ch[i] = a % 10; i++; a /= 10;
}
for (int k =0; k < i; k++)
{
if( !(ch[k] % 2))br++;
}
cout << br << endl;
}
```

Obviously both solutions are correct, but the second one is unnecessary complicated. A question arises here: should both solutions be assessed equally, or the first one should be given a bonus for better efficiency? But bear in mind that the second solution shows a bit more advanced programming skills.

The task above is a good example that supports Dijkstra, who says in [8] that it is imperative for a programmer, especially beginners, to spend sufficient time on developing algorithm with pen and paper, before entering code in computer.

Bloom's Taxonomy

Purpose of pedagogical taxonomies is to identify and position the educational goals of training, so that when implemented, they lead to a high-level thinking (higher rank). When a person has skills to think at a high level, he not only understands the nature of the studied objects and phenomena, but also discovers laws, principles and rules, reveals trends and future behaviour, participates in constructive, creative and artistic activities, consciously and wisely manages and controls their own individual and social behaviour.

Cognitive Bloom's taxonomy [17] is built as a hierarchy of six levels from memorizing and reproduction of the studied material to solving problems. Passing through those levels is necessary to re-evaluate the existing knowledge in order to build new combinations of pre-learned concepts, methods, procedures (courses of action), including proceeding to finding a new solution. The domain of this taxonomy is formation of intellectual skills. Originally, Bloom classified intellectual behaviours in six major categories: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation.

Since the original publication of this taxonomy to nowadays, a number of weaknesses and practical constraints have been found [18, 19]. In 2000 a group of researchers led by Anderson, published a revised version of Bloom's taxonomy [20]. In the revised version, knowledge is not one-dimensional, but has two: knowledge and cognitive process. Knowledge is the subject content and the cognitive process shows what should be done with the subject content. The cognitive processes reflect different forms of thinking and since thinking is an active process, it uses verb forms. This dimension has six categories, and in the original taxonomy, but they have been renamed and transformed [19]. Since knowledge can be the result or product of thinking, not a form of thinking, the original category Knowledge is replaced by Remember. The categories Comprehension, Application, Analysis and Evaluation are preserved, but in their verb forms: Understand, Apply, Analyze, Evaluate. According to Anderson et al. [20], in the process of creation, induction is more complex process than deduction. This is why the authors change the order of categories: from Synthesis – Evaluation to Evaluate – Create.

In introductory programming courses, the first three levels can be implemented by imperative rules. We should be aware that students do not develop skills to think at a high level if they: only memorize and reproduce information; or understand and explain the generally concepts and ideas; or apply information and rules in familiar situations only. At these levels even weaker students do well.

The last three, however, require active and conscious students' participation. A higher level thinking skills to a higher level develop when students are actively engaged in the process, analyze information in order to capture knowledge and relationships; evaluate their own decisions and the way they operate; create new ideas, products, and points of view. Thus, the high levels of thinking are related to the three categories of the cognitive process: Analyze, Evaluate and Create as per the revised Bloom's taxonomy (Figure 1).

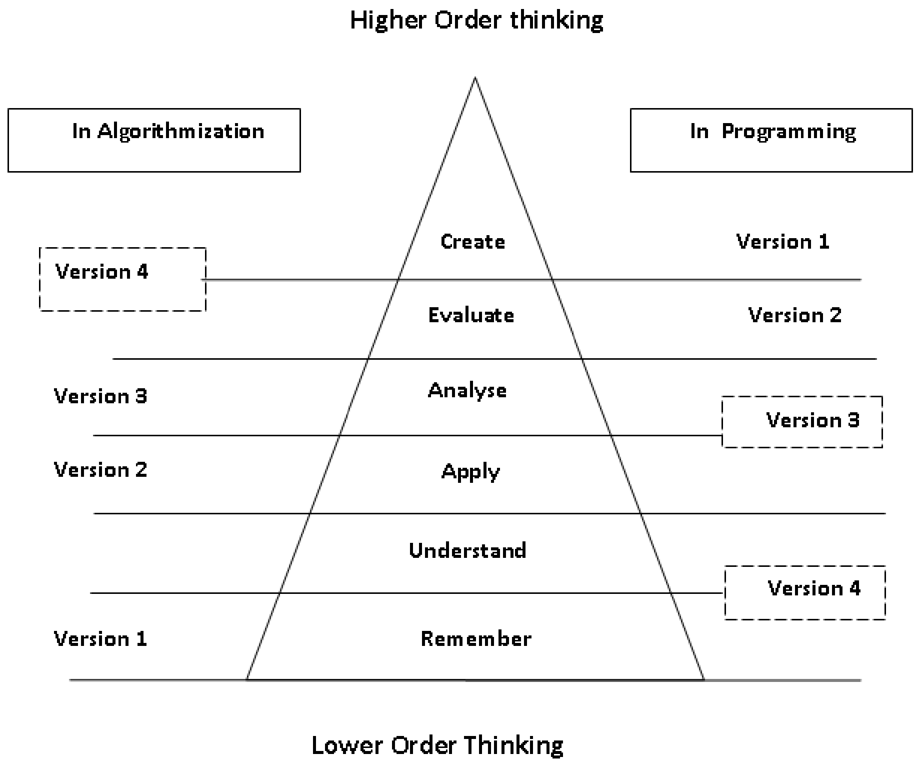

Figure 1: Revised Bloom's Taxonomy

It could be claimed that the Bloom's taxonomy describes well studying of programming. Indeed, we begin with composing source code, then go through correction of bugs, and finally we get correct programs. Following a certain style of programming, students could avoid some difficulties, which arise during development and modification of programs. The examples discussed in the previous section and their versions can be mapped easily to the levels of the revised Bloom's taxonomy. Even separate levels of the Bloom's pyramid can be discussed in the context of algorithmization and algorithm implementation, which in fact is the programming itself.

There is an inverse relationship between the level reached (effort made) with respect to the algorithmization and the level of mastering in programming, i.e., reaching a higher level in algorithmization allows a lower level of effort in programming.

Starting from level Remember (the definition of arithmetic mean) in algorithmization, which however requires level Create of the programming phase (implementation of loops) (*see Version 1*).

Continuing with the Apply level, which requires level Evaluate (optimization of loops) in the algorithm implementation (*see Version 2*).

Next follows level Analyze (formulas) requiring levels Analyze and Apply (removal of loop) in programming (*see Version 3*).

Finally it comes to Evaluate and Create levels of an algorithm that requires levels Remember (simple assignment operator) and Understand (explicit/implicit type conversion) of the stage of algorithm implementation (*see Version 4*).

Apparently, a common algorithm can solve a wide range of tasks, but getting in depth of the task in question, reveals that another algorithm, not that common, appears to be simpler and more effective

From the comments made here, it becomes clear that the main focus of the introductory programming course should be directed towards the algorithmization, which requires logical and mathematical culture. Based on a good algorithmization, it is easier to achieve good results in the implementation.

Conclusion

Studying Informatics requires development of methodic that not only helps students to grasp a large amount of knowledge, but also equip them with skills needed to master that knowledge, skills for independent acquisition of new knowledge and its critical understanding.

A methodical approach to building skills for critical thinking in the teaching programming is the implementation of various methods and approaches to solve a specific problem. Purposeful training for critical thinking allows students to overcome the reproduction level of understanding the course material, allows for getting deeper into the essence of emerging problems, and equip them with unconventional approaches to solve those problems.

Beginners in programming often underestimate role of the programming style and neglect it as something useless, which does not help solving problems. They often misunderstand the meaning of the teacher guidance for that while trying to focus on creating programs as soon as possible. As noted by Schorsch) [21], students often perceive the style of programming as something of secondary importance, non-integrated in the developing process of a program. They also easily forget the issue of programming style in their coursework.

The style of programming is one of the factors for creating qualitative software. According to Mohan and Gold [22], style of programming is one of the major problems in software maintenance. In software engineering, maintenance plays an important component of the software life cycle. The style of programming is part of maintenance, because a well-structured code helps maintainers to understand it easily. In contrast to that, beginners do a lot of mistakes and often write unreadable code, which is difficult to maintain. The personal teaching experience of the authors show that it is much easier to teach good style to students who have no prior programming experience and hopelessly hard to get students to adopt good programming style after they have adopted a bad one over years [16].

A simple program can work properly, even if it is written without style. The problems of bad programming skills emerge in larger programs or in situations, where the program need maintenance by another person, or migrated to another environment. Reusable software is an example of that.

References

[1] Sahoo *N.* C. & Chin J. G. W., 2010, *"An elaborate education of basic genetic programming using C++"*, Computer Applications in Engineering Education, 18 (3), pp. 434–448.

[2] Chakraborty, P., Saxena, P. C. & Katti C. P., 2013, *"A compiler-based toolkit to teach and learn finite automata",* Computer Applications in Engineering Education, 21(3), pp. 467–474.

[3] Deek, F., Kimmel, H. & McHugh, J. A., 1998, *"Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, Then Programming"*, Journal of Engineering Education, 87 (3), pp. 313–320.

[4] Gunther, J., Eames, B., & Nelson, D., 2011, *"Description of EduCOM: A graphical modeling and programming language for teaching and learning digital communication systems",* Computer Applications in Engineering Education, 19 (4), pp. 697–707.

[5] Fernández Leiva, A. J. & Civila Salas, A. C., 2013, *"Practices of advanced programming: Tradition versus innovation",* Computer Applications in Engineering Education, 21 (2), pp. 237–244.

[6] Atanov, G. A., 2004, *How to learn to use the knowledge, or Introduction to practice the activity of learning,* Donetsk (in Russian).

[7] Kalogeropoulos, N., Tzigounakis, I., Pavlatou, E. A. and Boudouvis, A. G., 2013, *"Computer-based assessment of student performance in programming courses",* Computer Applications in Engineering Education, 21 (4), pp. 671–683.

[8] Dijkstra, E. W., 1976, *A Discipline of Programming,* Prentice-Hall, Series in Automatic Computation.

[9] Beck, K. et al., 2001, *Manifesto for agile software development.*

[10] Wirth, N., 2002, Computing Science Education: The Road not Taken, ITiCSE Conference, Aarhus, Denmark, http://www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm (6/01/15).

[11] Govender, I., 2006, *Learning to Program, Learning to Teach Programming: Pre- and In-service Teachers' Experiences of an Object-oriented Language*, University of South Africa.

[12] Van Diepen, N., 2005, *"Elf redenen waarom programmeren zo moeilijk is (in English: Eleven reasons why programming is so difficult)"*, Tinfon, 14, pp. 105–107.

[13] Radošević, D., Orehovački, T. & Lovrenčić A., 2009, "*Verificator: Educational Tool for Learning Programming*", Informatics in Education, 8 (2), pp. 261–280.

[14] Van Tassel, D., 1978, Program Style, Design, Efficiency, Debugging, and Testing, 2 edition, Prentice Hall.

[15] Gal-Ezer, J. Vilner, T. & Zur, E., 2010, *Teaching Algorithm Efficiency at CS1 Level: A Different Approach, The Open University of Israel*, Tel-Aviv, Israel.

[16] Skūpienė, J., 2006, "*Programming Style – Part of Grading Scheme*", In Informatics Olympiads: Lithuanian Experience, ISSEP, pp. 545-552.

[17] Bloom, B., Engelhart, M., Furst, E., Hill, W. & Krathwohl D., 1956, *Taxonomy of educational objectives: the classification of educational goals,* Handbook I: Cognitive domain, New York: David McKay.

[18] Amer, A., 2006, "*Reflections on Bloom's revised taxonomy*", Electronic J. Res. Educ. Psychology, 4, pp. 213– 230.

[19] Krathwohl, D., 2002, *A revision of Bloom's taxonomy: an overview,* Theory into Practice, 41, pp. 212-218.

[20] Anderson, L. W., Krathwohl, D. R., Airasian, P.W., Cruikshank, K. A., Mayer, R.E., Pintrich, P.R., Raths, J. & Wittrock, M.C., 2000, *A taxonomy for learning teaching,* Boston, Allyn & Bacon.

[21] Schorsch, T., 1995, "*CAP: An automatic self-assessment tool to check Pascal programs for syntax, logic and style errors*",. Proceedings of the 26th SIGCSE technical symposium on Computer science education, USA, pp. 168-172.

[22] Mohan A., & Gold, N., 2004, "*Programming Style Changes in Evolving Source Code*", IEEE Proceedings of the 12th International Workshop on Program Comprehension, Bari, Italy, pp. 236–240.