Teaching Programming to Students in other Fields

Ivaylo Donchev
Department of Information Technologies
St Cyril and St Methodius University of Veliko Tarnovo
Veliko Tarnovo, Bulgaria

Emilia Todorova
Department of Information Technologies
St Cyril and St Methodius University of Veliko Tarnovo
Veliko Tarnovo, Bulgaria

***Abstract*—It is a fact that programming is difficult to learn. On the other hand, programming skills are essential for each program in the field of computing and must be covered in the curriculum, regardless of the profile. Our experience in the last 3-4 years shows a noticeable downward trend in students' results in computer science and similar programs. In this article, we comment on the reasons that have led to such a decline and we are looking for solutions by experimenting with motivated students from other areas of knowledge and comparing their progress in mastering basic concepts and mechanisms of programming with that of computer specialists.**

***Keywords*—Programming curricula; objects-first; teaching programming; object-oriented; education; programming; problem solving skills; politology**

I. Introduction

As a result of the modified admission model, there is already a significant number of students in the undergraduate programs at the Faculty of Mathematics and Informatics (FMI) at our university who do not have a good basis for studying the abstract matter of programming and the results shown are noticeably weaker than those of 5 or 6 years ago. Admission is now possible without a pre-selection through a math or informatics competition to ensure an acceptable level of problem solving skills and mathematical background. The courses thus formed are no longer a homogeneous group, and the differences in the level of knowledge and skills acquired and the potentialities, motivations and expectations of individual students are great. This challenge for teachers requires consideration and changes in the teaching methodology.

The lower level of applicants is mainly due to weaknesses in primary and secondary education. In different schools, even if they have the same profile, in the lessons of Informatics and Information Technology different material is studied, most often in line with the teacher's competences, and not with the pupils' specificities. Due to shortage of staff in education finding well-trained teachers is also a problem that is expected to deepen further in the coming years as there is a lack of interest at national level in programs preparing mathematics and computer science teachers.

Another negative factor for us is the tendency students from mathematical high schools, whose training is significantly better, to go to universities abroad believing that they will receive better education there. The motivation of a part of our students is only high incomes in the IT sector, without taking into account the necessary knowledge and skills to provide these incomes. Quite often this is accompanied by a misconception about IT technology – it is not uncommon for an IT professional to be considered a person who can install and customize an operating system and work with an office package. No account is taken of the fact that work with ready-made applications is not sufficient for a highly remunerated position.

Increasingly, computing in general and programming in particular are essential for students in other fields [1, p. 40]. Through them it is easy to develop critical thinking and problem solving skills that all students need to develop throughout their undergraduate career. Despite the accumulated more than 60 years of experience, however, teaching programming is still considered quite a challenge [11, p. 111] especially with regard to introductory courses [12]. Many researchers refer to learning programming as extremely difficult activity [13], [14]. Our faculty traditionally provides training of students from other faculties in elective and facultative disciplines related to informatics and information technologies. Our observation, in particular, of our longstanding work with students from Politology undergraduate program held by the Faculty of Philosophy shows that they are smart, literate and disciplined and can learn almost everything if it is properly presented to them. Believing that programming may be useful for students from other areas of study who wish to use it as a tool in cross-disciplinary work [1, p. 42], we decided to experimentally teach programming in the eighth semester (of eight semesters) in two consecutive academic years (2015-2016 and 2016-2017), with the consent of both students and teaching department.

Our hypothesis is that by solving simple, carefully selected practical tasks using pure object-oriented language, it is possible for a limited number of lessons and more extracurricular work to acquire fundamental procedural and object-oriented programming concepts including data types, control structures, functions, objects, classes, inheritance, and polymorphism, as well as design concepts and principles like abstraction, decomposition, encapsulation and information hiding, separation of behavior and implementation. The experience from this experiment will help us to decide how to reorganize our CS1 and CS2 courses to improve the knowledge and results of our students at the Faculty of Mathematics and Informatics (FMI).

The article is further structured as follows. In Section II we review the model of training in the introductory courses in computer science programs at our faculty. In Section III we argue the changes we have made in the Computer Technology course in order to be able to compare the achievements of Politology students with those of Informatics students as well

as the teaching methods. In Section IV we analyze the results of experimental training. In the conclusion, we point out the possibilities for further development of the experiment and draw conclusions that experimental training of non-specialists can help in improving the training of our faculty students.

II. Introductory Programming Courses

The bachelor degree programs in the field of computing at FMI are: Informatics, Computer Science and Software Engineering. In [2], the six main implementation strategies for introductory courses are described, covering the first two years of training that are current today: imperative-first, objects-first, functional-first, breadth-first, algorithms-first and hardware-first. We apply the traditional imperative-first model with three main courses, which sequence, workload and content vary from one program to another (see Table I).

The Informatics and Computer Science programs use the C++ hybrid language for introductory courses, which until now we considered to be a good choice for several reasons:

- In secondary schools this was the most commonly used language and the students had experience with it;
- The language is powerful enough for both procedural and object-oriented programming;
- Using a hybrid language is easier to make paradigm shift from procedural to object-oriented programming;
- Students had a good foundation and coped with the heavier C++ syntax, including pointer manipulation and memory management;
- The availability of good textbooks and C++ tools in the language of our country.

Objects are introduced at the latest to Informatics students. The course of ADS (Algorithms and Data Structures) here is entirely procedurally implemented. Computer Science program introduces the OOP (Object-Oriented Programming) concepts in the second semester and the ADS course uses them actively. The algorithms and abstract data structures in the STL library are also considered here. In the fourth semester a second language is added: Java. For both programs in the 4th semester a new paradigm – declarative, is also studied (logic and functional programming).

Our newest bachelor degree program is Software Engineering. It is developed in cooperation with IT business and its curriculum is strongly influenced by its specific needs. It started in 2016-2017 academic year. Here the first language is the pure object-oriented C#, but again the procedural concepts are first studied. The OOP course in the second semester is complemented by **.**NET Web Development, which uses the same development environment and the same language (C#). In the third semester a second language (optionally and a third) is introduced. The ADS course comes late in the 4th semester and relies on the already well-trained C#. The training is complemented by UML, JavaScript, Logic and Functional Programming Courses, Android Mobile Apps.

TABLE I. CS1-CS2 Programming Curricula

Program	Course / Language	Workload (lectures/practice)	semester
Informatics	CS1: Foundations of programming (C++)	45/45	1
	CS1: Algorithms and Data Structures (C++)	45/45	2
	CS2: Object-Oriented Programming (C++)	45/45	4
	CS2: Logic Programming (Prolog)	30/15	4
	CS2: Functional Programming (Lisp/Haskell)	15/15	4
Computer Science	CS1: Foundations of programming (C++)	30/30	1
	CS1: Programming in C++ (C++)	30/30	2
	CS2: Algorithms and Data Structures (C++)	30/45	3
	CS2: Programming in Java (Java)	30/30	4
	CS2: Nonprocedural programming (Prolog, Lisp)	30/30	4
Software Engineering	CS1: Introduction to Programing (C#)	30/60	1
	CS1: Object-Oriented Programming (C#)	30/30	2
	CS2: Web Programming with **.**NET (C#)	15/45	2
	Elective C++ / PHP	15/30	3
	Elective Java / Event-driven programming (C#)	15/30	3
	CS2: Algorithms and Data Structures (C#)	30/45	4
	Elective UML / Mobile Android Applications	15/30	4
	Elective JavaScript / Logic and Functional Programming	15/30	4

For Software Engineering, it is still early to draw conclusions, but in the other two programs as a result of the above-mentioned problem with the changed kind of our students, this model already shows weaknesses and the curricula need reconstruction. This of course is relevant not only to the introductory courses.

III. Modified Course in Computer Technology

The elective course “Computer Technology” for the Politology undergraduate program is in the 8th semester and has 30 academic hours of lectures and 30 hours of laboratory lessons – a normal amount for an introductory course in programming. In our case, however, we wanted to compare students’ achievements with those of their Informatics

colleagues who passed the three CS1 courses. It is clear that even if the last did not make enough effort, with such a difference in the workload, Informatics students have a great advantage. If we consider the fact that following the imperative-first approach with parallel running courses on Operating Systems and Fundamentals of Informatics, they learn well the mechanics of running, testing and debugging and have a clear idea of how a computer is running the program, how the processor works, what is happening in memory, their advantage becomes even greater – there is evidence to support the thesis that students who have inaccurate and incomplete understanding of the process of implementing a program face greater difficulties in their learning [3]. An additional advantage for Informatics students is the study of methods of program verification according to the methodology described in [15]. Since this methodology relies on good mathematical knowledge, we are limited with nonspecialists (Politology students) only to testing with Visual Studio tools. It makes sense to compare the achievements of the two groups of students only by some criteria. We have chosen these to be the degree of perception of object-oriented concepts and their application in practice in the design and implementation of program solutions from a familiar to students' problem area. A natural choice of approach in this case is the object-first approach that emphasizes the early use of objects and object-oriented design. This avoids focusing on the syntax of the programming language and the details of procedural constructs implementation. A pure object-oriented language is appropriate for such an approach. We chose C# in order to be able to apply our experience in Software Engineering bachelor program as well as to compare the difficulties encountered in the two groups. The development environment is the same for all students – Visual Studio. Though this is an industrial integrated development environment, its code editing features, including code completion, parameter info, quick info, and member lists, are extremely helpful in learning syntax and avoiding errors (both syntactic and logical). The environment encourages writing good code, has the ability to generate code from UML diagrams, to refactor and analyze code. In addition, through this environment students can touch on important aspects of real-world software development.

It should be noted that objects-first is not a well-defined term [6]. Different authors have their own understanding of this concept. It is our understanding that from the very beginning students have to get a clear idea of the essence of the two concepts of this paradigm – object and class, to distinguish between them, to find suitable for modeling classes and objects in the problem area, to discover and present the relations between them. Therefore, the first lecture is purely theoretical and is focused on object-oriented analysis and design.

Our course develops knowledge and skills from the Software Development Fundamentals (SDF), Programming Languages (PL) and Software Engineering (SE) knowledge areas, taught in direct relationship to C# language constructs. In order to solve practical tasks, little knowledge of Algorithms and Complexity (AC) is needed, focusing on the use of ready library implementations – search, sort, select. We will mention that the SDF knowledge area differs from the old Programming Fundamentals form CC2001 [2]. It focuses on the entire software development process, including algorithms and data structures and basic software development methods and tools.

Given a limited number of hours and our desire to develop practical skills, lectures do not run in their typical format. Along with the presented theoretical material, code and diagrams are loaded in the development environment. The lecturer develops in live examples, runs, debugs, modifies, refactors, demonstrates how to use environment tools, analyzes the quality of code (commented on automatically calculated code metrics). During lectures practical skills are acquired, not only theoretical knowledge. Students can ask questions at any time. The examples shown in the lectures are then further developed in the laboratory sessions and variations of them are given for homework tasks.

The sequence of topics is as follows:

1) Overview lecture on object-oriented technology: The object model – foundations, major elements of this model (Abstraction, Encapsulation, Modularity, Hierarchy); Classes and objects – state, behavior and identity of the object; operations with objects, roles and responsibilities; relations between classes.

2) Working with variables, operators and expressions – statements, identifiers, primitive data types, arithmetic operators, assignment.

3) Creating and managing classes and objects – encapsulating, defining and using classes, access control; defining methods (parameters, parameter passing by reference and by value, out parameters, default value, named arguments).

4) Using decision statements (if, switch) and iteration statements (while, for, do).

5) Constructors and destructors: Predefined constructors; Garbage collection; Static methods and data.

6) Properties (read-only, write-only, auto): Partial class definitions; Anonymous types; Refactoring.

7) Values and references: Nullable types; the class System.Object; Organization of memory; Boxing and unboxing; safe conversion of types.

8) Structures: Enumerations; Arrays; Generic types.

9) Collections: List, Dictionary; Collection initializers, find methods, predicates, and lambda expressions; Querying in-memory data using query expressions (LINQ – selecting, filtering, ordering).

10)Inheritance: Declaring a derived class; Calling constructors of a base class; Assignments between objects in class hierarchy.

11)Virtual methods: Polymorphism.

12)Theoretical lecture: Classification; the Importance of Proper Classification; Identifying Classes and Objects; Key Abstractions and Mechanisms.

13)Managing: Errors and exceptions.

14)Interfaces: Definition, implementation, referencing a class via interface, explicit implementation, implementation of multiple interfaces.

15)Abstract classes: Sealed classes and methods.

Two of the lectures are highly theoretical and fully language independent (1 and 12). The first introduces the object-oriented technology and the second comes after the students have gained practical experience in order to summarize the lessons learned and to provide guidance for its proper implementation.

The difference with the object-oriented course for Software Engineering students is that no overloading, extension methods, reflection, indexers, delegates, events are included here. These language capabilities, although very useful, are not key to object-oriented technology. Their lack is not a problem for the experiment, as the results will be compared to those of Informatics students who study C++ (the implementation of such concepts is on a radically different basis). The test tasks are selected so as not to imply the use of these concepts.

Although our approach is objects-first, in order to be able to solve practical problems, and students to understand the code generated by the environment or written by the instructor, some fundamental programming concepts common to all paradigms are also introduced early, although not in details. These are variables and primitive data types, expressions and assignments, conditional and iterative control structures, functions and parameter passing.

The laboratory works are mostly focused on the use of C# features and their application for the implementation of object-oriented models. We start with .NET and Visual Studio 2015 environment. The examples are an important part of the course for both labs and lectures. In an object-first curriculum, the objects presented by the instructor play a key role in motivating and explaining an object-oriented approach [4]. For this reason, we chose tasks from everyday life, games and such related to students' work on case studies in their program (Politology). The first examples of exercises are semi-finished projects in which students must add functionality – a method, property, or change object behavior by modifying an already implemented functionality. In this way students immediately see the outcome of their work on the code, and this works motivatingly. In the examples we try to show good programming practices without commenting that these are classic programming patterns, going into details about their nature and summarizing the situations in which they are applied. We rely on students to build an intuitive notion. We took some ideas of tasks of those presented at the Nifty Assignments session at the annual SIGCSE meeting [5]. Some examples are developed and expanded into several lectures and exercises, and in the process of study of a new concept or mechanism the implementation changes (and sometimes the overall design). Such is the example of the card game discussed in [7].

When starting a new task, first, with the help of the instructor, object-oriented analysis of the problem area is made, the classes and objects are designed, the use cases are examined, which helps to clarify the roles and responsibilities of the objects. To document ideas and design, we use a lightweight and often informal UML notation. We rely, especially in the initial exercises, on wizards to create classes and their components, including those made from class diagrams. This helps a lot in avoiding syntax errors and learning a good style of writing and structuring code. Another very useful feature of Visual Studio is Code Snippets that are designed and used by professionals as a means of speeding up code writing, but in our case the benefit of them in combination with code completion, parameter info, quick info, and member lists was rather in the direction of learning the syntax of the language and avoiding syntax errors.

In the laboratory work we apply pair programming, which has long been used in industry [8] and is increasingly applied in training. Research has shown that it improves both code quality and efficiency of student pairs compared to individual work [9], [10]. This was also useful in extracurricular work, which we relied on to compensate for the smaller number of lesson for lectures and exercises compared to this for the informatics students. Learning to work in pairs, discussing tasks, tracking the work of their partner, changing their roles, have worked well in developing homework projects in teams of 2 or 3 students.

IV. Analysis of Learning Outcomes

Prior to conducting this experimental training, we conducted pedagogical studies with students from FMI to track the degree of mastering of key concepts and mechanisms in programming. Informatics and Computer Science students from upper classes have also been observed, that is a delayed check. The results show a good and comparatively persistent level of proficiency in procedural concepts, but more problems with object-oriented design. The concepts of a class and an object are perceived almost entirely from a language point of view – the class is a user-defined data type, and the object is a variable of that type. Definition of individual classes and implementation of their methods according to a predetermined exact specification do not hinder students (about 80% of them), but there are serious difficulties in creating an adequate object-oriented model of a problem area, finding the necessary classes and the exact relationships between them, building communication between objects (only 30% do well with this task). We attribute these results to the imperative-first model applied to the training of these students.

In order not to stress the Politology students with multiple tests and quizzes, in the experimental training we did not conduct a three-stage classical pedagogical experiment to formally compare their achievements with those of Informatics students. Due to the small number involved, the statistical processing of data from such an experiment would not yield reliable results. Instead of this, we gathered empirical data from teachers' notes of activity during the sessions, achievements, difficulties encountered by each student, results of homeworks, a test and a final test, the same for the "specialists" (Informatics undergraduate program who play the role of the control group) and "non-specialists" (4th year Politology students). Both tests involved solving a task on a computer.

The results shown in the final test (on one and the same problems) are similar to the average result with a slight lead in favor of the Politology students (Table II), but here we have to keep in mind that the test was so prepared as to cover only the material that they know best.

TABLE II. TEST RESULTS

Grade	Politology		Informatics
	First Test	Final Test	Final Test
A	16%	18%	23%
B	46%	46%	23%
C	22%	20%	12%
D	12%	14%	27%
F	4%	2%	15%

It is noteworthy the low number of poor assessments of Politology students, as well as the rather high number of F and D grades for the Informatics students. There is no clearly expressed average level among the second – the largest groups are those of excellent and poor marks. We believe that this is due to the free admission to the program. Some of the undergraduates drop out after the second year and look for a job elsewhere. In the group of Politology students, predominate estimates B, with the percentage of A being also high (18% of the final test). Poor scores are a minimum – 2 on the first test and 1 on the second. We attribute this to the efforts all made and their responsible approach to the tasks.

Our observations of lectures and exercises, teamwork on projects, and analysis of scripts we can determine as common difficulties for "nonspecialists", which lead to the bound up with them mistakes in logic and design, the following:

- Passing of parameters to a function – ref and out parameters.
- Constructing more complex Boolean expressions.
- They do not understand the essence of the task and from there they cannot judge when to use static components of the classes.
- They do not detect quickly when polymorphism is suitable for use, but start solving using conditional logic.
- Casting and type conversions (including boxing and unboxing).
- Using an interface as a type.

Compared to the Software Engineering students who study the same language, nonspecialists find it more difficult to use procedural constructs of C#. This is natural, as this is an introductory course for them. Thanks to the Visual Studio environment, we have reduced not only syntax errors but also some other often noticed in the past – defining a variable from a missing class, calling a missing method, a field of undefined or inaccessible type, an attempt to access private components. The environment immediately notifies of such situations.

Politology students are dealing better with design and, in particular, with choosing the right relationships between related classes. Informatics students often confuse and generally prefer inheritance as a key concept for object oriented programming without exploring the possibility of implementation with a lighter mechanism. Inheritance is often used only as a means of achieving re-usability of code without the presence of true "is-a" relationship between classes. Informaticians in turn are better at working with generic types and algorithms.

Looking for success indicators, we conducted a poll about the school the students graduated from, their results in Maths, Informatics, Information Technologies, as well as average grade from school and average grade at the university. For the specialists, we also studied their exam results at our faculty. An indisputable indicator of success in studying programming for both groups of students has been good Maths results. Maths skills help to cope with the high degree of abstraction of the material studied and the building of proper models and algorithms. Among Politology students, there were also some who graduated from mathematical high schools and some of the concepts (mainly procedural) studied there were familiar to them. It was easier for them to build upon their old knowledge. High results in programming have been shown by students who have high average grades at the university. This shows that programming is not that difficult if one is ready to make efforts. For the Informaticians expectedly a high correlation between the results in Fundamentals of Informatics, Fundamentals of Programming and Object-Oriented Programming was found.

V. CONCLUSION

The results achieved give us reasons to believe that the experimental training was successful. Politology students have mastered enough of the basic concepts and acquired practical programming skills. It will be interesting to track if some of them will change their profession and with additional qualification in the master degree to effectuate in the field of software development.

In the future, we are preparing to expand the experiment with formal statistical processing of accumulated empirical data, and to include experimental training in "Operating Systems" of students from the newly created hybrid faculties of Applied Linguistics and IT and History and IT, and also to compare the achievements with those of Informatics students.

As long as we cannot influence the quality of our students' selection, we will try to apply the accumulated teaching experience and successful methods of training non-specialists to the introductory courses of the computer specialties where, as we have discussed above, in recent years, we have very heterogeneous in interests and potential students.

The first conclusion is that it is necessary to replace the C++ language with C# and go entirely to objects-first approach. We expect this to bring good results for weaker students once it has worked with non-professionals. C# is easier for first language, especially for object oriented programming.

Efforts should also be made towards motivating students' learning and their greater engagement in the learning process. This is possible by selecting examples and solving tasks, working on semi-finished projects so that the results of the work are immediately visible. We will give up all mathematical tasks and focus on more entertaining samples for students, like examples in [5].

Another useful technique that we will apply is programming in pairs in lab sessions and working in small teams on homework projects. In the experiment, the Politology

students' teamwork helped weaker members of the teams to gain confidence and fill the gaps in their knowledge.

In spite of the sufficient number of academic hours for programming in computer programs, we believe that it will be beneficial to engage students with more extracurricular activities including work on a course project from the first year of study.

REFERENCES

[1] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, Computer Science Curricula 2013. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, ACM, New York, 2013

[2] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, Computing Curricula 2001. Computer Science. Final Report, http://www.acm.org/education/education/education/curric_vols/cc2001.pdf

[3] Ma, L., Ferguson, J., Roper, M., Wood, M., Investigating and improving the models of programming concepts held by novice programmers, Computer Science Education 21 (1), pp. 57 – 80, 2011

[4] Hummel, J., Caspersen, M., Alphonce, C., Hansen. St., Bergin, J., Heliotis, J., Kölling, M., Nifty Objects for CS0 and CS1, ACM SIGCSE Bulletin - SIGCSE 08 Volume 40 Issue 1, March 2008, pp. 437-438, ACM New York

[5] Nifty Assignments, http://nifty.stanford.edu/

[6] Bennedsen, J., Caspersen, M., Model-Driven Programming, in Reflections on the Teaching of Programming, LNCS 4821, pp. 116–129, 2008, Springer-Verlag, Berlin, Heidelberg, 2008

[7] Sharp, J., Microsoft Visual C# 2013 Step by Step (Step by Step Developer), Microsoft Press, 2013

[8] Cockburn, A., Williams, L., The Costs and Benefits of Pair Programming, Extreme Programming Examined, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001, pp. 223–243

[9] McDowell, Ch., Werner, L., Bullock, H., Fernald, J., The Effects of Pair-Programming on Performance in an Introductory Programming Course, In Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE '02), Cincinnati, Kentucky — February 27 – March 03, ACM, New York, 2002, pp. 38–42

[10] Radermacher, A., Walia, G., Investigating the Effective Implementation of Pair Programming: An Empirical Investigation. In: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11), Dallas, TX, USA — March 09 - 12, 2011, ACM New York, 2011, pp. 655–660

[11] Caspersen, M., Bennedsen, J. (2007). Instructional design of a programming course: a learning theoretic approach. In: Proceedings of the Third International Workshop on Computing Education Research, Atlanta, Georgia, USA, pp. 111–122.

[12] Meyer B. (2004) The Outside-In Method of Teaching Introductory Programming. In: Broy M., Zamulin A.V. (eds) Perspectives of System Informatics. PSI 2003. Lecture Notes in Computer Science, vol 2890. Springer, Berlin, Heidelberg, pp. 66–78.

[13] Simon, S., Fincher, S., Robins, A., Baker, B, Box, I., Cutts, Q., de Raadt., M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., Tutty, J. (2006). Predictors of success in a first programming course. In: ACM International Conference Proceeding Series; Proceedings of the 8th Australian conference on Computing Education, Hobart, Tasmania, Australia, pp. 189–196.

[14] Gomes, A., Mendes, A. (2007). Learning to program – difficulties and solutions. In: Proceedings of the 2007 International Convergence on Engineering Education, Coimbra, Portugal.

[15] Todorova. M., Applying Program Verification Methods in Software Specialists Education, 7th International Technology, Education and Development Conference, Valencia, Spain, 2013, pp. 6260-6270.