Informatics in Education, 2008, Vol. 7, No. 2, 159–172
© 2008 *Institute of Mathematics and Informatics, Vilnius*

Object-Oriented Programming in Bulgarian Universities’ Informatics and Computer Science Curricula

Ivaylo DONCHEV, Emilia TODOROVA
Department of Information Technologies, Veliko Tarnovo University
G. Kozarev str. 3, 5000 Veliko Turnovo, Bulgaria
e-mail: i.donchev@abv.bg, emilia_todorova@yahoo.co.uk

Received: July 2008

Abstract. Teaching object-oriented programming (OOP) is related to many difficulties. There is no single view on their causes among the university teachers. The results of applying various methods of teaching – with early or late introduction of the objects, are controversial too.

This work presents the results of a study designed to analyze and classify the difficulties encountered in the teaching of OOP in Bulgarian universities as well as the possibilities for dealing with them. Two viewpoints have been considered – of lecturers and of students. The issues under consideration are: when and what should be studied, what should be stressed, what languages and environments should be used, what examples are the most suitable, and what educational goals the programming courses should achieve.

Our investigation was aimed also to confirm or cast aside our suppositions that important aspects in teaching/learning OOP are being underestimated: great attention is being paid to the data in a class at the expence of the behavior of the objects in a program; more than necessary is being stressed onto the syntactic peculiarities in defining classes and objects without detailed clarification why they are needed; the auxiliary didactic tools that are being used are insufficient.

Keywords: object-oriented programming, OOP paradigm, procedural paradigm, curriculum, pedagogy.

1. Introduction

Object-oriented programming (OOP) became the most powerful programming paradigm in the last twenty years. It is widely used in education and industry. In the curricula of all universities, educating students in the fields of Informatics and Computer Science OOP is always presented. There is no doubt how useful is OOP. Teaching it covers all important concepts like well-structured programming, modularity, program design, as well as techniques for solving problems that just recently found place in computing curricula: team work, large systems maintenance, and software reusability.

Yet teaching and learning OOP and programming in general remains a difficult task (Börstler and Sharp, 2003; Ebrahimi and Schweikert, 2006; Kölling, 1999a). To answer the question why it is so, we should first clear up its role and place in the university computing curriculum. This work presents the results of an investigation of the computing curricula and syllabus of 12 bulgarian universities, educating students in the fields of Informatics and Computer Science, and two separate inquiries: 27 university lecturers having between 1 and 26 years of experience in teaching programming and OOP, and 214 students in Informatics and Computer Science between 1st and 4th year of study (bachelor degree). The aim of this investigation was to outline the difficulties in learning OOP and find the reasons for them.

2. When to Study OOP?

This is one of the most discussed questions (Lister and Berglund, 2006). Historically, the procedural paradigm precedes the object-oriented, and until twenty years ago OOP was presented in the last courses of study, only for outstanding students. Object orientation was considered mostly a superstructure of the procedural paradigm, not as an independent one (Burton and Bruhn, 2003). Later, more and more universities began to include OOP as an introductory course in programming. The main reason for this is the so-called "paradigm shift". Getting used to write programs in object-oriented style seems very hard for students after they had got used to program in procedural (Liu *et al.*, 1992). Students master easily the principles of object orientation if OOP is included as the first course in programming. It appears that the difficulty is in switching over the paradigms, not in learning OOP itself (Kölling, 1999a).

The extreme followers of this idea consider that the introductory course in programming should be a course of problem solving, and problem solving can be mastered using the object-oriented paradigm. The training in OOP has to start with modeling and early introduction to the basic concepts ("objects-early"). Modeling can be performed by the means of a traditional programming language and subsidiary tools for visualization.

On the other hand, the opinion that OOP is extremely difficult for beginners because of its high level of abstraction compared to procedural programming, is widely spread (Ebrahimi and Schweikert, 2006; Hadjerrouit, 1999). Most lecturers in computer science agree that abstract thinking is a critical component in learning informatics and programming in particular (Bennedsen and Caspersen, 2006). Starting to learn programming with OOP not always gives good results. In (McCracken and Almstrum, 2001) the results of a investigation are shown, according to which 30% of the students in Great Britain and the USA, who had started programming with OOP, did not understand the basic concepts of objects. Many questions arise in a combined approach: to include in one course elements from both paradigms stating that the procedural and object-oriented programming are not naturally excluding each other paradigms. When designing complex systems of objects, the first phase of building up objects and the links among them is always followed by the phase of their implementation, and the latter needs good knowledge of structural programming and algorithms (Cecchi and Crescenzi, 2003). We think it is possible to include concepts like classes, methods and inheritance in the introductory programming course, but it seems to be unsuitable to teach object-oriented design (OOD) without previous experience in using classes and hierarchy of classes.

The results of our investigation show that over 81% of the inquired lecturers consider that OOP should be taught after an introductory course in procedural programming. The opinions divide 3 : 2 on the question if the object orientation should be studied before the course in Algorithms and Data Structures (ASD). The other controversial question – which model is more successful – "programming-first" or "design-first", gives favor (85%) to programming, and this result corresponds to the statement of the final report "Computing Curricula 2001" of IEEE and ACM (The Joint Task Force on Computing Curricula, 2001), that in near future the model "programming-first" will remain prevailing. The group of questions, aimed to clarify the place of OOP in curricula plainly shows that it is not among the introductory courses. The arguments of the inquired teachers are that this paradigm is of higher level of abstraction and that the transition from procedural to OOP is easier and more natural (81%).

The results from students' inquiry to a great extent coincide with those of lecturers': 22% had great difficulties learning OOP, 65% met some difficulties, and only 13% had no difficulties (Fig. 1).

As for the students is it hard to outline the reasons for the difficulties they met, we included in their inquiry 5 of the most frequently pointed by the lecturers cathegories of difficulties. The students had to choose at least one of them. As the most common reason for difficulties, the students point (39%) the high level of abstraction, which is followed by the complexity of the programming language used (27% – Fig. 2).

For the lack of enough experience and knowledge, it is clearly that the students find as more difficult one the question "Which transition is more complex: from procedural to object-oriented programming or the opposite?". In full consciousness of the fact that their answers can not be indicative, we included this question only in the inquiry for 3rd and 4th year students, who had already passed an introductory course in programming, a course in OOP and an elective course in Java or C#. Our aim was to compare the opinion of the future programmers with that of the lecturers. For the same reason we included in the students' inquiry the question when to study OOP: 58% from the inquired consider that both transitions are equally difficult, and 30% consider that the first is easier. (It is interesting that, according to students, the needed period of time to conclude this transition is 5–6 months, while lecturers give 2–3 months for this.) The inquiries show different results of students' and lecturers' opinion about when OOP to be studied. Students state (57%)

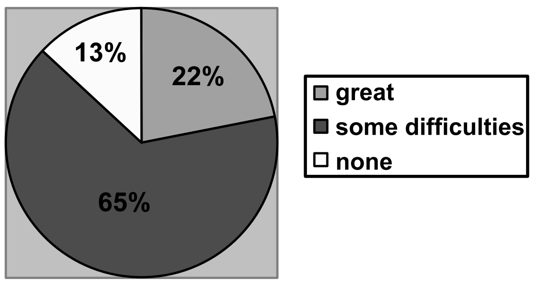

Fig. 1. Students' inquiry results: Rate of the difficulties met when learning OOP.

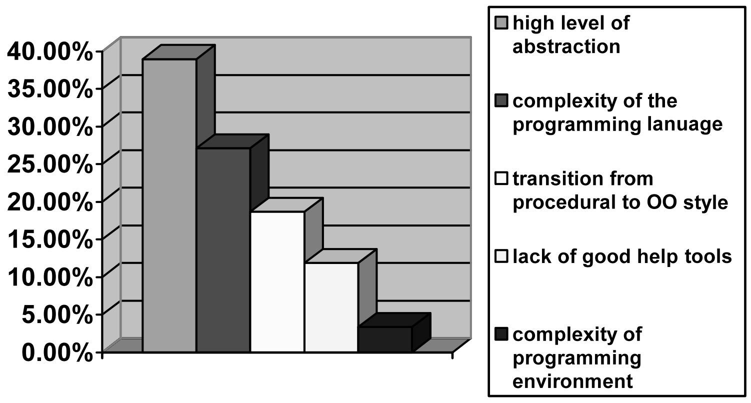

Fig. 2. Main reasons for the difficulties in studying OOP, according to students.

that OOP should be studied in the beginning of the programming courses in the curricula; 25% prefer to study it before ASD. This seems a bit strange if we consider the results of the question “Does experience with procedural programming help you to learn OOP?” 75% of students say “yes”. Very similar are the results from lecturers’ inquiry: 81% state that experience with procedural programming helps learning OOP. Taking into account the fact that 81% of the lecturers (probably the same persons) consider that the transition from procedural to OOP is easier and more natural, we can conclude that OOP should be preceded by a course in procedural programming in order to obtain better understanding and easier transition to the OO paradigm. We suppose that the different results in students’ and lecturers’ answers are caused by the insufficient number of lectures and labs in OOP. (42% of the students consider the number of lectures insufficient, 84% consider the number of labs insufficient.) A positive moment is that the students understand the importance of the studied concepts and style of programming for their future work as IT specialists: 74% state they will use OOP approach in the future.

3. Where Should We Put the Accents on Teaching/Learning OOP?

To answer properly this question we should first clear up what is the object-oriented model and object-oriented approach. OOP has a long history, but yet there are many definitions of object orientation.

Rentch (1982) defines OOP as inheritance, encapsulation, methods and messages, as in Smalltalk. Pascoe (1986), similarly, considers object-oriented terminology as a Smalltalk perspective. He defines object-oriented approach in the terms of encapsulation, abstraction of data, methods, messages, inheritance and dynamic linking. According to him, some languages that do not provide one or two of these features, can not be named “object-oriented”, and gives as an example Ada, which does not support inheritance.

Other authors like Robson (1981), and Thomas (1989) lay the emphasis on the idea of passing messages among objects and dynamic linking as fundamental for OOP. These authors are, too, under the influence of Smalltalk. On the other hand, Stroustrup (1987) states that OOP may be viewed as programming with using inheritance and that message passing is just an applied technique but not a component of the paradigm.

Nygaard (1986) discusses OOP in terms and concepts of the objects in Simula. With this language, the execution of a program is represented as cooperative work of a set of objects. This set as a whole simulates objects from the real world. Objects that have common features compose a class.

Lastly, Wegner (1987) examines the object-oriented approach in the terms of objects, classes, and inheritance. Objects are autonomous entities that have a state and respond to messages. Classes arrange objects by their basic attributes and operations. Inheritance serves to classify classes by their shared commonality. Then

Object orientation = objects + classes + inheritance

The last definition (Wegner, 1987) for object-oriented approach is the most widely accepted (McGregor and Korson, 1990; Lindsey and Hoffman, 1997).

As Bertrand Meyer (2000) notes, “Object-oriented is not a boolean condition” and we can not dogmatically define criteria for object orienattion. He emphasizes that “Not everyone will need all of the properties all the time” and proposes the following full list of object characteristics:

- Classes: The method and the language should have the notion of class as their central concept. Classes should be the only modules. Every type should be based on a class.
- Assertions: The language should make it possible to equip a class and its features with assertions (preconditions, postconditions and invariants), relying on tools to produce documentation out of these assertions and, optionally, monitor them at run time.
- Feature-based computation: Feature call should be the primary computational mechanism.
- Information hiding: It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.
- Exception handling: The language should provide a mechanism to recover from unexpected abnormal situations.
- Static typing: A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.
- Genericity: It should be possible to write classes with formal generic parameters representing arbitrary types.
- Single inheritance: It should be possible to define a class as inheriting from another.
- Multiple inheritance: It should be possible for a class to inherit from as many others as necessary, with an adequate mechanism for disambiguating name clashes.
- Repeated inheritance: Precise rules should govern the fate of features under repeated inheritance, allowing developers to choose, separately for each repeatedly inherited feature, between sharing and replication.

- Constrained genericity: The genericity mechanism should support the constrained form of genericity.
- Redefinition: It should be possible to redefine the specification, signature and implementation of an inherited feature.
- Polymorphism: It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.
- Dynamic binding: Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.
- Run-time type interrogation: It should be possible to determine at run time whether the type of an object conforms to a statically given type.
- Deferred features and classes: It should be possible to write a class or a feature as deferred, that is to say specified but not fully implemented.
- Memory management and garbage collection: The language should make safe automatic memory management possible, and the implementation should provide an automatic memory manager taking care of garbage collection.

The results of our inquiry show that most of the lecturers abstain from giving formal definition of OOP. They prefer the following approach: students construct themselves the idea of OOP during the course. Lecturers outline inheritance, encapsulation, classes and objects as the most important features of OOP. The summary about OOP characteristics, pointed by the lecturers as important is given in Table 1.

One can see that the lecturers had united some concept into one topic. Such are the concepts of a class and an object. This is due to the strong relationship between them and most of the specialists state that they can not be studied separately. In OOP objects are the primary units used to create abstract models, while a class is an abstract description of a set of objects. Exactly this very strong relation is the reason students often to mismatch thase concepts. To avoid this misconception, the lecturer should emphasize that a class is a specifcation of an set of objects, it is not the actual object. In technical terms, a class defines a new type in the system.

Table 1

Summary of the results of the answers about the importance of studied concepts and their level of difficulty

OOP concepts	Important, according to % of the lecturers	Rate of difficulty (1–10), according to students
Inheritance	85%	3.69
Encapsulation	66%	5.57
Classes and objects	51%	3.11
Polymorphism	51%	6.92
Dynamic linking, virtual methods, abstract classes	33%	7.32
Abstract data types	33%	5.66
Operator overloading	18%	5.37

One can see that the concepts dynamic linking, virtual methods, and abstract classes are united as well. The reasons are again the strong relationships among them – virtual methods can not exist without a mechanisnm for dynamic linking, and abstract classes contain virtial methods by definition. Besides, these three concepts are presented in one chapter in most of the popular in Bulgaria textbooks, by example (Todorova, 2001).

We note that among the most important concepts determining characteristics are not presented, like: interaction and communication among the objects in a program, relations among classes, operations with objects, exception handling as well as specific mechanisms, which are not peculiar to all OO languages – multiple inheritance and parametrized types (templates in C++).

The absence of some concepts in Table 1 can be explained by the fact that topics like relations among classes and object interaction are discussed in the course of object-oriented analysis and design (OOAD) and for the limited number of lectures and labs they very rarely find place in the course in OOP. Not all examined computing curricula have a separate course in OOAD, and if there is such, as a rule it is after the course in OOP. Thus we state that these aspects should be emphasized in the courses in programming.

Our opinion is confirmed by the results of our inquiry: the lecturers had to evaluate the importance of the four suggested for example construction. They outline as important for teaching of programming the following:

- simplicity and elegance of construction, as we remember from classic Pascal of Wirth – 85%;
- short examples, consisting of few pages of code, which nevertheless illustrate the power of OO ideas – 66%;
- classes with rich contents and abilities, easy to use – 51%;
- a set of classes having good interaction among them – 33%.

One can see that 33% of them pointed interaction among classes, whish is not presented in Table 1. Besides, the lecturers agree that the accent in teaching OOP should fall on the OOD and overall understanding of the approach (66% of all and 100% of those, who answered the question). They also agree that the introductory courses in programming should emphasize on both conceptual understanding and algorithmic problems. These results confirm our opinion that exactly attaining proficiency in OOD is the main factor to surmount most of the difficulties in learning OOP.

The analysis of our inquiry showed the following difficulties in learning OOP:

1. Misunderstanding the difference between a class and an object and the relations between them.
2. Difficulties in building up and understanding the links among classes and their interaction.
3. Difficulties in building an overall understanding how the program solving the given problem works.
4. Difficulties in testing and debugging of programs, reading someone else's code and detecting of logic errors.
5. Misunderstanding of memory operations – copying constructor, destructor, assignment operations, etc.

6. Difficulties in using heavy professional programming environments.
7. Lack of good help tools.
8. Students write code with a low rate of reusability.

In our opinion the crucial reasons for these difficulties are connected with abstract way of thinking, what is a basic component in studying programming. To avoid problems in this respect we should not underestimate mastering data types and structures in earlier courses. Procedural programming and OOP are not mutually excluding each other paradigms and it is admissible to include elements from OOP in an earlier course aiming at easier transition to OOP in future.

Difficulties 1, 2, 3 and 8 are directly connected with the design; difficulty 5 is specific for C++, and 4, 6 and 7 are related to programming in general.

We believe, the design and overall understanding of the approach are determinant for successful study of OOP, and a teaching course should stress on assimilation of the main concepts of data abstraction, encapsulation, inheritance, and polymorphism.

We should note the fact that almost half (43% of the students) prefer to learn writing code and language specific details in the course of OOP. We suppose this is due to their wish to gain quick professional recognition.

4. Which Language and Environment are Appropriate?

Köling (1999a; 1999b) makes a deep analysis of these problems. He states that difficulties in studying OOP come not from OOP itself, but from the available instruments and the rather complex languages and environments in use. Another important conclusion is that programming languages are not good or bad as a whole, they are good or bad for specific purposes. Many textbooks use procedural programming as a pathway to object concepts. Köling states that one of the main influences in this is the hybrid language C++ and its popularity. C++ was developed as an extension to C, so many people can consider that object orientation is just another language construction but not a basic paradigm.

The choice of language strongly depends on the method chosen: early or late introducing to objects. The opinion is that in the case of early introduction it is better to use a pure object-oriented language, e.g., Eiffel or Java. In the other case, when OOP is studied after an introductory course in procedural programming, the logical choice is a hybrid language as C++. This choice strongly depends on the development environment. For the needs of learning it should be easy to use, should have integrated help tools, should not stress very much on the user's interface (Kölling, 1999b). The language itself should support all important OOP concepts. It is still arguable if multiple inheritance is such a concept, or may be left for a later stage in the curriculum, after a course in modeling.

The results of our inquiry show that the most used language for introduction to OOP in bulgarian universities is C++. Only two from twelve universities, teaching students in informatics and computer science, have obligatory introductory courses in Programming with Java. All the rest use C++ in the introductory course, as well for OOP. Later, as an advanced course students can choose between Java and C#. Only a few lecturers use

C# or Delphi. The reason for this is not only the popularity of the language, but also the place of OOP course in the curriculum: after an introductory course in procedural programming. So the transition to OOP appears not to be so difficult. There is no united opinion about which language and environment are most appropriate for studying OOP. The answers include Borland Delphi, C#, Java, and C++.

The above results differ from the statistics shown on Fig. 3 (Labelle, 2006), but we must keep in mind that none of the used programming languages is indispensably the best for teaching/learning. Anyway, Java and C++ are favorites in our inquiry as well.

One of the interesting results of our inquiry is that lectures rarely use auxiliary tools in the process of teaching. Mostly used are multimedia projectors for demonstrations of ready working projects. Only 18% point the lack of auxiliary tools as a reason for the difficulties; complexity of languages (33%) and development environments (30%) are also pointed as such reasons.

Students' answers to the question which language is appropriate for OOP are influenced by the studied in their universities languages: 43% can not estimate which language is the best for OOP, 57% of those, who answered the question, point C++, and 20% point Java. Similar results we got on the question "What language and environment do you prefer to use during the course?": 65% prefer Visual C++, and 20% – Borland Java – i.e., what is used at the moment. The results of our inquiry and our observatoins show that the importance of the used in the teaching process programming environment is considerably underestimated by the lecturers in Bulgaria. Very often we use heavy professional environments like Microsoft Visual Studio and Borland CodeGear RAD Studio. Working with these environments additionally increases the difficulties and frightens the begginers in programming. On the other hand, there are lecturers, using common text editors and

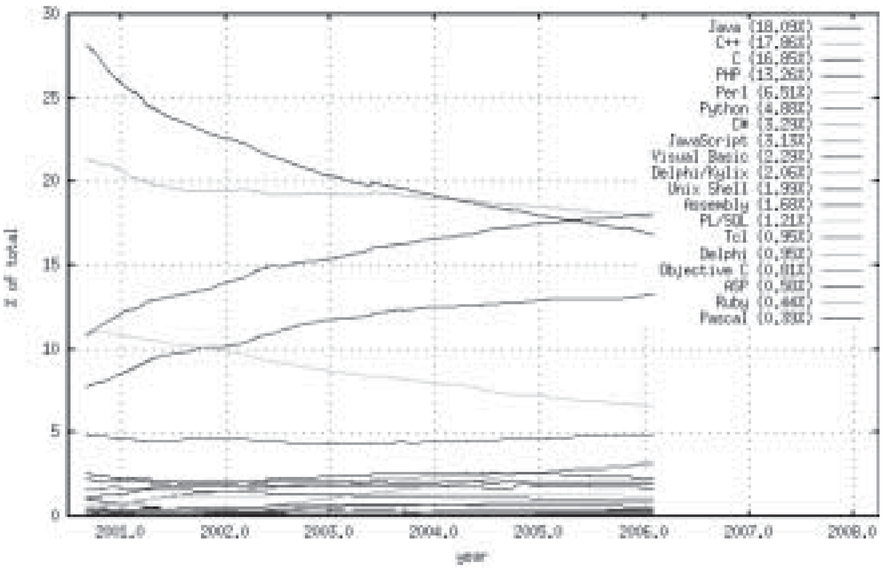

Fig. 3. Programming language usage graph (Sept. 2000–Feb. 2006).

command line compilers. The main reason for this is the lack of specific environments, developed especialy for the needs of teaching OOP in C++. The situation is better for Java, where the popular BlueJ and Karel J Robot are successfully used.

5. What Examples Are To Be Used in the Course?

Bruce (2005) cites the results of a psychological research, according to which people learn easier if at the beginning of the study they are given particular examples, and general conclusions are made later (inductive method). That is why we assume that choosing proper examples plays a very important role in acquiring the concepts of OOP. On this topic, our inquiry gave the following results: Teachers prefer short examples, including classes with rich contents and behavior, easy to use and modify. Some lecturers use fragments from bigger projects. Much attention is paid to the interaction and the links among classes. Many of the difficulties occured to a great extent are connected with the used by the greater part of the inquired lecturers (and students, respectively) language C++. Non-assimilated concepts about memory manipulation are pointed as main reasons for this: copy constructors, virtual methods, etc. – 66%. When students do not understand relations among classes, they can not construct an overall picture of how programs work and how the problems are solved. As another reason the lecturers point not well mastered work with pointers (33%). For this reason all inquired lecturers include pointers as members of example classes. All mentioned above contributes to the integral understanding of the paradigm. Almost all the inquired lecturers use more examples modeling objects from "the real world" and less examples solving pure mathematical problems.

Lecturers' answers, as well as students' answers, undoubtedly outline the following problems: diffuculties in debugging of programs, reading someone else's code, finding logic errors. Therefore it is reasonable to recommend putting the accent on these aspects as early as possible – in the introductory course in programming.

The students, too, show preference for examples "from the real world": 57%. The second place is hold by half-ready big projects, needing to be enriched by functionalities (32%). Pure mathematical examples are favorites of 11% of the inquired students.

6. What Is the Sequence in Teaching/Learning Programming and What Are the Basic Educational Tasks?

The last group of questions in lecturers' inquiry aimed to outline the stages of studying programming and the purposes it should achieve. The inquired were asked to number (with priority, the highest is 1), the proposed minimum number of stages, as well as to add new ones in their judgment. As the inquired gave very heterogeneous answers we assume that the received generalized results are not indicative and therefore are not suitable to invent an applicable sequence for the practice: most of the suggested stages are parallel to a great extent. The results can be more realistically interpreted as pointing out the most important elements which should be included in the teaching/learning process:

- learning language syntax;
- language constructs;
- writing code;
- learning to solve problems;
- developing a way of thinking;
- gaining new skills.

A certain sequence of studied topics entirelly depends on the model of teaching/learning that is chosen. Having in mind the fact that in Bulgaria it is the model "programming-first" in general, and the most of the courses follow the strategy of late introduction to objects, we assume that the recommended model by Computing Curricula 2001 (The Joint Task Force on Computing Curricula, 2001) is most suitable.

The results of our inquiry about the educational goals, set in the courses in programming, do not give a solid base for interpretation. The lecturers suggested quite difficult by formulatoin goals, which was hard to combine into groups. The statistic processing was additionally hardened by the small number of inquired lesturers, but in Bulgaria it is the prevailing part of the people, teaching programming (and especially OOP) in universities. Although, we think it is possible to outline the following educational goals:

- to give students substantial grounding in programming;
- to engage their attention and provoke motivation for self-dependent work;
- to build up creative skills and pleasure from programming;
- to master concepts via abstraction;
- to integrate technologies;
- to build up algorithmic way of thinking.

7. Conclusion

The results of our research confirm the view that important aspects of OOP teaching are underestimated:

1. In most teaching courses the emphasis is laid on the data described in the class at the expense of objects' behavior and interaction in the program.
2. Teaching courses usually are directed to the syntactic features of constructing classes and objects in a concrete programming language without giving the students the opportunity to achieve knowledge why these classes and objects are necessary.
3. We lack the experience of practical study and knowledge about the full cycle of designing, introducing and maintaining software.
4. Coordination of curricula concerning programming learning and software design is not sufficient.
5. The auxiliary didactic tools that are being used are insufficient.

All the teachers we inquired understand the difficulties and main reasons for them but this fact does not lead to proper changes in teaching courses. The aim of an OOP course is to build up an object oriented thinking of the students. That is why it is better constructions connected with the specific language to be taught after discussing upon some common paradigm concepts and mechanisms. The relationships and communications among objects but not the algorithms should take central place in an OOP course.

The opinion of the inquired teachers could be boiled down to the following conclusions:

1. It is difficult to the students to learn OOP as an introductory course in programming. Basic reason for that is the higher level of abstraction in comparison to the one of the procedural style.
2. The OOP concepts are better assimilated by students with experience in procedural programming. Therefore it is more appropriate to teach OOP after an introductory course in procedural programming.
3. It is not reasonable a formal definition of OOP to be stated/formulated. Students have to come to the core of the paradigm by themselves.
4. The course should stress on the OOD and overall understanding of the approach.
5. Most of the teachers prefer short examples, including classes with rich contents and behavior, easy to use and modify. Examples connected with modeling objects taken out of "the real world" are more suitable than solving pure mathematical problems.
6. Most of the students can not "read" someone else's code as well as detect logical mistakes and write proper comments;
7. The program code written by the students is of a lower level of reusability which in our opinion is again due to some voids in students' design skills.

References

Bennedsen, J. and Caspersen, M. (2006). Abstraction ability as an indicator of success for learning object-oriented programming. *ACM SIGCSE Bulletin*, **38**(2), 39–43.

Börstler, J. and Sharp, H. (2003). Learning and teaching object technology, Editorial. *Computer Science Education*, **13**(4), 243–247.

Bruce, K. (2005). Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. *ACM SIGCSE Bulletin*, **37**(2), 111–117.

Burton, Ph. and Bruhn, R. (2003). Teaching programming in the OOP era. *ACM SIGCSE Bulletin*, **35**(2), 111–114.

Cecchi, L. and Crescenzi, P. (2003). C : C++ = JavaMM : Java. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java* (ACM International Conference Proceeding Series), vol. 42. Kilkenny City, Ireland, 75–78.

Decker, R. and Hirshfield, S. (1994). The top 10 reasons why object-oriented programming can't be taught in CS 1. In *Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education*. Phoenix, Arizona, United States, 51–55.

Ebrahimi, A. and Schweikert, C. (2006), Empirical study of novice programming with plans and objects. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. Bologna, Italy, 52–54.

Groven, A. and Hegna, H. (2003). *OO Learning, a Modeling Approach*. `http://prog.vub.ac.be/~imichiel/ecoop2003/workshop/papers_for_presentation/15-Groven-Hegna-Smordal.pdf`

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*. Cracow, Poland, 171–174.

Kölling, M. (1999a). The problem of teaching object-oriented programming, Part 1: Languages. *Journal of Object-Oriented Programming*, **11**(8), 8–15

Kölling, M. (1999b). The problem of teaching object-oriented programming, Part 2: Environments. *Journal of Object-Oriented Programming*, **11**(9), 6–12

Labelle, F. (2006). *Programming Language Usage Graph*. `http://www.cs.berkeley.edu/~flab/languages.html`

Lindsey, A. and Hoffman, P. (1997). Bridging traditional and object technologies: Creating transitional applications. *IBM Systems Journal*, **36**(1), 32–49.

Lister, R. and Berglund, A. (2006). Research perspectives on the objects-early debate. In *Annual Joint Conference Integrating Technology into Computer Science Education*. Bologna, Italy, 146–165.

Liu, C., Goetze, S. and Glynn, B. (1992). What contributes to successful object-oriented learning? In *OOPSLA '92 Proceedings*, 77–86.

McCracken, M. and Almstrum, V. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. December 01, 2001, Canterbury, UK.

McGregor, J. and Korson, T. (1990). Introduction – object-oriented design. *Communications of the ACM*, **33**(9), 38.

Meyer, B. (2000). *Object-Oriented Software Construction*. Second Edition. Prentice Hall PTR.

Nygaard, K. (1986). Basic concepts in object oriented programming. *ACM SIGPLAN Notices*, **21**(10), 128–132.

Pascoe, G. (1986). A. Elements of object-oriented programming. *Byte*, **11**(8), 139–144.

Rentsch, T. (1982). Object-oriented programming. *ACM SIGPLAN Notices*, **17**(9), 51–57.

Robson, D. (1981). Object-oriented Software Systems. *Byte*, **6**(8), 74–86.

Sprague, P. and Schahczenski, C. (2002). Abstraction the key to CS1. *Journal of Computing Sciences in Colleges*, **17**(3), 211–218.

Stroustrup, B. (1987). What is Object-Oriented Programming? *Lecture Notes in Computer Science*, **276**, 51–70.

The Joint Task Force on Computing Curricula (2001). *Computing Curricula 2001* (final report), December 2001. `http://www.computer.org/education/cc2001/final`

Thomas, D. (1989). What's in an Object, *Byte*, **14**(3), 231–240.

Todorova, M. (2001). *Programming in C++*. SIELA, Sofia.

Wegner, P. (1987). Dimensions of object-based language design. *ACM SIGPLAN Notices*, **22**(12), 168–182.

I. Donchev recieved his MCs degree in informatics from Veliko Tarnovo University. Currently he is principal lecturer at Veliko Tarnovo University and is working towards his PhD degree in computer science education. His research is focused on methodology of teaching object-oriented programming and object-oriented analysis and design. He is a member of the John Atanasoff Society of Automatics and Informatics (SAI).

E. Todorova is a graduate of Sankt Petersburg State Electrotechnical University (LETI), 1987, Russian Federation. She is a MSc in computer engineering. She received her PhD at the same university, in 1993. Since 1995 she is an assistant professor, and since 2003 – an associated professor and head of the department of Information Technologies at Veliko Turnovo University, Bulgaria.

Objektinis programavimas Bulgarijos universitetų informatikos ir informacinių technologijų programose

Ivaylo DONCHEV, Emilia TODOROVA

Objektinio programavimo mokymas yra gana sunkus dalykas. Aukštosios mokyklos neturi vieningo požiūrio, kodėl taip yra. Rezultatai, gauti naudojant įvairius mokymo metodus – anksčiau ar vėliau pradedant mokyti objektų – taip pat yra gana prieštaringi.

Šiame straipsnyje pateikiami tyrimo rezultatai, skirti sunkumų, iškilusių mokant objektinio programavimo Bulgarijos universitetuose, analizei, klasifikavimui ir sprendimo būdams nagrinėti. Visa tai aptariama iš dviejų pozicijų: dėstytojų ir studentų. Pagrindinis dėmesys skiriamas objektinio programavimo mokymo problemoms nagrinėti: kada ir ko reikėtų mokyti, kas svarbiausia, kokias kalbas ir aplinkas reikėtų naudoti, kokie pavyzdžiai būtų labiausiai tinkami, kokius mokymo tikslus programavimo kursai turėtų pasiekti.