Designing and Deploying Programming Courses: strategies, tools, difficulties and pedagogy

Stelios Xinogalos

Abstract
Designing and deploying programming courses is undoubtedly a challenging task. In this paper, an attempt to analyze important aspects of a sequence of two courses on imperative-procedural and object-oriented programming in a non-CS majors Department is made. This analysis is based on a questionnaire filled in by fifty students in a voluntary basis. The issues of the programming courses that are investigated refer to: the *strategy* selected for the introduction to programming; the sequence of the *programming techniques and languages* taught and the transition from the one to the other; students' *difficulties* with programming in general and with imperative-procedural and object-oriented programming in specific; the *teaching* and *learning design* of both courses; and the *material* that students rely on for learning programming. Based on the analysis of students' replies on the questionnaire, related work and the instructor's experience on teaching the courses, conclusions are drawn regarding all the aforementioned aspects of designing and deploying programming courses. The main contribution of the paper is the fact that all the important and interrelated aspects of a sequence of two programming courses are investigated in conjunction, providing realistic implications and guidelines for improving the quality and effectiveness of existing programming courses and designing and deploying new courses. The main results refer to the usage of a pseudo-language for an introduction to programming, the transition from procedural to object-oriented programming, the intrinsic difficulties of learning programming, and practices for a more successful teaching and learning design of programming courses.
Keywords: Programming course design; teaching and learning programming; procedural programming; object-oriented programming; pedagogy

INTRODUCTION

Teaching and learning programming is accompanied with many problems, such as program design, complexity of programming language features and novices' fragile knowledge (Robins et al., 2003). In this sense, designing programming courses for undergraduate students has always been a challenging task (Xinogalos, 2012a). Crucial and informed decisions have to be made from the very beginning at least for the following issues:

- Selecting a *strategy* for an initial approach to teaching programming.
- Selecting a *programming language* that supports the selected strategy and meets the goals of the course and the program of studies.
- Selecting one or more *programming environments*.
- Selecting a general *teaching* and *learning design*.
- Selecting *textbooks* and preparing *educational material*.

Selecting a strategy for an initial approach to teaching programming is the first decision that has to be made. Several choices are available and some of them have gained widespread acknowledgment, while alternative ones that have not gained such widespread acceptance have also been proposed in the literature. The basic strategies are the *imperative-first*, *functional-first* and *objects-first*, while one of the best-known alternative approaches is the *model-first approach* (Bennedsen & Caspersen, 2004). The imperative-first and functional-first strategies were heavily used for decades, while the objects-first strategy attracted teachers' attention the last decade. For a long period of time extended research was carried out regarding the best choice of strategy for an introduction to programming with main opponents the imperative-first and objects-first strategy. Although, the results of the relevant studies are contradictory the majority of researchers seem to agree that students face more difficulties during their transition from imperative-procedural programming to object-oriented programming and not vice versa (Decker & Hirshfield, 1994; Hadjerrouit, 1998 & 1999; Tempte, 1991; Wick, 1995). Some of the difficulties faced by students - with prior experience on an imperative-procedural language – during their introduction to OOP are the following: although the OO problem solving technique is considered more natural, it demands a new way of thinking that cannot be easily acquired by students with experience on problem solving with a procedural language (Tempte, 1991); students find it difficult to use correctly OOP concepts and tend to treat methods as procedures, ignoring their role in OOP (Handjerrouit, 1998; Handjerrouit, 1999). On the other hand, other researchers state that object-oriented languages demand knowledge of basic programming structures and characteristics and capabilities prior to using an OOP language (Cooper et al., 2003). It is obvious that deciding what strategy to rely on for the introduction to programming is not an easy and straightforward decision. Moreover, in the case of a series of programming courses decisions have to be made regarding the overall strategies, or else what programming techniques will be taught and with what sequence.

The next step is s*electing a programming language* that supports the selected strategy and meets the goals of the course and the program of studies. The available programming languages are numerous and selecting the one that will be used is a multi-criteria decision. Researchers have proposed lists of criteria (Parker et al., 2006), key features (McIver and Conway, 1996) and suggestions (Kaplan, 2010) for supporting teachers in selecting the first programming language. One of the most extensive list of criteria has been proposed by Parker et al. (2006) and includes the following criteria: software cost; programming language acceptance in academia; programming language industry penetration; software characteristics; student-friendly features; language pedagogical features; language intent; language design; language paradigm; language support and required training;

and student experience. McIver and Conway (1996) go a step beyond these usual considerations proposed as selection criteria and stress out key features for evaluating a potential teaching language that refer to its syntax, semantics, error diagnostics and generally its "virtues" as a teaching language.

Selecting one or more programming environments usually follows the selection of the programming language. A wide variety of programming environments are available that fall into three main categories: *professional programming environments*, *educational programming environments* (Georgantaki & Retalis, 2007) and *programming microworlds* (Brusilovksy et al., 1997). Each category has its own advantages and disadvantages. For example, microworlds are considered by several researchers and instructors ideal for introducing novices to programming, but on the other hand there might be some hesitation whether the knowledge acquired in the context of a microworld is transferred to the conventional programming language used afterwards. Educational programming environments are targeted to novices, and just like microworlds, are simple and user-friendly and incorporate various forms of educational technology: software visualization, program animation, structure editors and so on. Moreover, educational programming environments can incorporate conventional programming languages or a subset of a conventional language. Professional programming environments are targeted to professionals and offer an abundance of tools that are useful for a programmer, but at the same time confuse novices.

Another important decision refers to the overall *teaching and learning design*. Several questions have to be answered, with the most important ones being the following: What will the proportion of lecture and lab sessions be? What approaches will be used for teaching during lectures and labs? What types of assessment of students' knowledge will be utilized? How will students be supported in studying and what material will they be provided with? How can we motivate and engage students in a programming course?

Finally, *textbooks* have to be selected and *educational material* has to be prepared based on the teaching and learning design of the course. Preparing educational material usually demands a lot of effort and time and requires continuous evaluation of its didactical effectiveness. Programming is a cognitively demanding area and although it is generally important for everyone in today's knowledge society, students' interest has decreased. Preparing high quality educational material requires a deep knowledge of the extended literature on students' difficulties and misconceptions, hands-on experience on teaching programming and continuous search for contemporary innovative teaching and learning approaches with the aim of motivating and engaging students in learning.

It is clear that in several occasions the decisions for the aforementioned issues are influenced by factors not related directly to pedagogy. Available human and technical resources, as well as instructors' prior experience on specific programming languages influence more or less the decisions made.

In this paper, an attempt to study the aforementioned issues regarding the design and deployment of programming courses from the viewpoint of the teacher and the students is made. Specifically, students' replies on a questionnaire regarding the following aspects of a sequence of two programming courses in a Technology Management Department are analyzed: sequence of programming techniques and languages/environments; general difficulties in learning programming; difficulties with imperative-procedural and object-oriented programming concepts/constructs; teaching and learning design of programming courses; educational material. The main goal of this study was to evaluate the overall design and deployment of a sequence of two programming courses, as well as students' difficulties and attitudes in order to provide insights and guidelines for course designers and teachers struggling to deliver more successful programming courses. The rest of the paper is organized as follows. In Section 2, the questions and the methodology of the research carried out are presented. The results of the questionnaire are analyzed in Section 3. Finally, section 4 concludes with insights and guidelines for designing and deploying programming courses.

THE STUDY

In this section important data regarding the study carried out are presented. First of all, the study took place at a Technology Management Department where studies last for four academic years. The academic year is divided in two terms, or else in two semesters lasting 13 weeks each. This department offers two compulsory programming courses at the 2nd and 3rd semester. The first course is "Computer Programming" and introduces students to the main programming concepts and constructs using the imperative-procedural programming technique and C as a programming language. The second course is "Object-Oriented Design and Programming" and introduces students to the object oriented programming technique, using Java as the programming language. The courses were designed back in 2004 when the Department was established. The sequence of learning units, the programming environments utilized, the textbooks and the educational material designed and developed were heavily based on the extended research carried out regarding teaching and learning programming to novices and especially imperative-procedural programming. For the "Object-Oriented Design and Programming" course extended research was also carried out in the context of the course and several reformations were made in it. Critical information for both courses will be presented in conjunction with the results of the study.

The study took place during the academic year 2011-2012 when the sequence of the two programming courses had come to a point where a good overall design seemed to have been accomplished. In order to validate our beliefs and choices a decision to investigate students' opinions was taken. Our goal was to investigate the following questions:

Question 1: How difficult are for students the introduction to imperative-procedural programming and the transition to object-oriented programming? How effective are the measures taken for supporting students in their difficulties?
As will become clear later the sequence of programming techniques was in a high degree a one-way road. However, based on the extended literature in the subject we anticipated difficulties both for the introduction to imperative-procedural programming and afterwards the transition to OOP. Measures for these difficulties have been taken and our goal was to investigate both how severe the aforementioned difficulties and how effective the measures taken are.

Question 2: What are students' main difficulties in learning programming independently of the programming technique utilized?
Learning programming encompasses various processes, such as designing algorithms and transferring them to a programming language, learning the syntax of the language, using programming environments and so on. Our aim was to see what students' main difficulties in the context of the specific courses are, in order to see whether students need further support in some of these higher level processes.

Question 3: How difficult is each imperative-procedural and object-oriented programming concept/construct for students to comprehend?
It is clear that extended literature is nowadays available regarding students' difficulties both with imperative-procedural and object-oriented programming concepts/constructs. As a matter of fact, this literature was taken into account when designing the courses and the educational material, as we have already mentioned. So, in this study our aim was not to investigate what students' difficulties are, but whether our choices regarding the time allocated for the various learning units and the special measures (such as special libraries and educational environments) taken for dealing with known difficulties were correct.

Question 4: What do students think of the various aspects of the teaching and learning design applied in the programming courses?

Question 5: What kind of material do students find more helpful for learning programming?

In order to investigate the aforementioned questions, data were collected from a specially designed questionnaire consisting of 73 closed-type, Likert-scale questions (1=not at all, 2=slightly, 3=averagely, 4=much, 5=very much). The questions were grouped in the following categories:

- Sequence of programming techniques and languages (Table 1: 6 questions).
- General difficulties in learning programming (Table 2: 13 questions).
- Difficulties with imperative-procedural programming concepts/constructs (Table 3: 15 questions).
- Difficulties with OOP concepts/constructs (Table 5: 20 questions).
- Teaching programming (Table 7: 6 questions).
- Learning programming (Table 8: 7 questions).
- Material for study (Table 9: 6 questions).

Several of the questions presented in Tables 2, 3, 8 and 9 were based on the survey used in a study carried out by Lahtinen et al. (2005) regarding the difficulties faced by novice programmers. The survey used by Lahtinen et al. (2005) included 30 items. Some of these items were substituted in our study by more questions, as described in detail in the corresponding sections. Moreover, the analysis of students' replies is presented in the context of the available literature in the field.
The questionnaire was filled in on a voluntary basis after the final exams of the 3rd semester "Object-Oriented Design and Programming" course based on Java. Students had also attended the 2nd semester "Computer Programming" course based on C. Fifty students out of sixty eight that attended the exams, filled in the questionnaire (73.5% of students participated in the study). The distribution of the 50 students that participated in the study based on their semester of studies (within the 8 semester bachelor degree) is the following: 29 students having finished the 3rd semester, 5 students the 5th semester, 5 students the 7th semester and 11 students the 9th semester. Several of these students had failed on the exams of the "Computer Programming" course and also several had failed on the exams of the "Object-Oriented Design and Programming" course. Specifically, the 5th semester students had attended the "Object-Oriented Design and Programming" course the previous academic year, the 7th semester students two academic years ago and the 9th semester students 3 academic years ago. All these students had either failed on the final exams or did not take the exams at all.
For each question of each category the following statistics were calculated and are presented in the corresponding Tables in order to facilitate interested readers in evaluating the results in alternative ways:

- *percentage* of student's replies falling into each one of the 5 possible answers
- *mean* and *standard deviation*
- *confidence value* computed with an alpha of 0.05 that indicates a 95% confidence level and provides a confidence interval that is *mean* ± *confidence value*.

ANALYSIS OF THE SURVEY RESULTS

Sequence of programming techniques and languages

The strategy for the introductory programming course was heavily based on *students' prior experience in procedural programming*. The majority of first-year students, approximately 90% every year, have prior experience in procedural programming. Specifically, these students are introduced to the main principles of procedural-imperative programming (variables, control structures, subprograms and arrays) at their last year in Secondary Educations and are examined in this material for entering in Tertiary Education. This introduction to procedural programming takes place with a simplified subset of Pascal translated in students' mother tongue. Our *hypothesis* was that it would be easier for students to *build on their prior knowledge and acquire a deeper understanding of fundamental programming concepts using a real imperative programming language*. So, the imperative-first (procedural) strategy was selected for the first programming course.

The selection of the programming language was based on criteria proposed in the literature - such as *acceptance in academia* and *penetration in industry* – and factors specific to the Departments' *curriculum* that gives emphasis on digital systems and technologies. For the aforementioned reasons, C was considered a good choice for the introductory programming course. However, it was clear from the very beginning that C is not the best choice for first programming language in terms of its pedagogical features and special attention should be paid on designing the course and supporting students in comprehending and utilizing the fundamental programming concepts.

The second programming course is based on the object-oriented programming technique and Java. Object-oriented design is considered important and is utilized in subsequent courses on analysis and design of Information Systems. Moreover, Java is a popular language supporting various types of applications, as well as other courses on the department's curriculum. For example, Java is used for network programming in the context of the Distributed Systems course. Students' prior introduction to programming with the imperative programming technique and C was expected to make the transition to OOP quite difficult according to the relevant literature. This was taken into account and special attention was paid in designing the course. For example, the educational IDE BlueJ was used and the didactical material was designed taking into account the extended research on students' difficulties and misconceptions when introduced to OOP. Despite this fact, experience and rigorous evaluation on teaching the specific sequence of courses for two consecutive years showed that the transition from procedural to OOP was difficult for our students. For this reason, we decided to use the programming microworld objectKarel just in the beginning of the course for a more straightforward and smoother transition to OOP.

In order to evaluate this sequence of programming techniques and languages as a whole and test our hypotheses we posed students with the closed type questions presented along with the corresponding results in Table 1. The most important results are analyzed in the following paragraphs.

Table 1. Sequence of programming techniques.

	Question	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T1.1	How much did the use of a pseudo-language help you in your introduction to programming?	43	3.81	1.3	4	0.39	7	12	14	28	39
T1.2	How easy was the comprehension of the corresponding concepts in the programming language C?	43	3.72	0.83	4	0.25	0	5	37	39	19
T1.3	How much more difficult do you believe the introduction to programming would be directly with the programming language C?	43	3.7	1.1	4	0.33	5	7	30	30	28
T1.4	How difficult was the introduction to procedural programming with C?	48	2.85	0.99	3	0.28	10	21	46	19	4
T1.5	How difficult was the transition from procedural programming (C) to OOP (Java)?	48	3.19	1.04	3	0.29	4	21	40	23	12
T1.6	How much were you supported in comprehending the technique and the fundamental concepts of OOP by the use objectKarel at the beginning of the OOP course?	48	2.96	1.13	3	0.32	6	38	19	29	8

Introducing novices to programming with a pseudo-language, instead of a real programming language, is preferable.
As we have already mentioned the vast majority of the students (86%) that participated in the study had been introduced to imperative-procedural programming with a pseudo-language in Secondary Education. This introduction to programming with the pseudo-language is considered important by the majority of students. Specifically, students reported the following regarding the introduction to programming with a pseudo-language:

- 67% of the students reported that the pseudo-language helped them much or very much in their introduction to programming (T1.1: Mean=3.81, Std. Dev.=1.3)
- nearly 9 out of 10 students believe that the introduction to programming with a real programming language, namely C, would be more difficult (T1.3: Mean=3.7, Std. Dev.=1.1). To be more precise, 58% of the students believe that this introduction to programming would be much or very much and 30% averagely more difficult with a real programming language instead of the pseudo-language.
- what might be considered even more important is that 58% of the students believe that they did not just comprehend easier programming concepts in the pseudo-language teaching, but also comprehended them with much or very much easiness in the real programming language afterwards (T1.2: Mean=3.72, Std. Dev.=0.83).

The introduction to procedural programming with C is quite difficult for students (T1.4: Mean=2.85, Std. Dev.=0.99).
Nearly half the students reported that faced an average level of difficulty (46%), while one fifth of them (23%) reported that their introduction to procedural programming with C was significantly or even very difficult. The aforementioned results refer to all the students. The corresponding results for the two groups of students – with (43 students) or without (5 students) prior experience with a pseudo-language – are different: the mean for students with prior experience is 2.74 and for for students withour prior experience 3.8. This fact confirms the aforementioned results regarding the support provided to novices when using a pseudo-language for introducing them to programming.

The transition from procedural programming to OOP seems to be even more difficult than the introduction to procedural programming (T1.5: Mean=3.19, Std. Dev.=1.04).
One fourth of the students (25%) reported slight or no difficulty at all; an important number of students reported an average level of difficulty (40%); and more than one third of students (35%) reported that the transition from procedural programming to OOP was significantly or even very difficult. The corresponding percentage of students that reported having significant or very much difficulty with the introduction to procedural programming was 23%. This result was not surprising, since several researchers have found that students face more difficulties during their transition from imperative-procedural programming to object-oriented programming and not vice versa (Decker & Hirshfield, 1994; Hadjerrouit, 1998 & 1999; Tempte, 1991; Wick, 1995). As a matter of fact, these studies combined with our experience on teaching the specific sequence of courses for two consecutive years was the reason for using objectKarel at the beginning of the OOP course for a more straightforward and smoother transition to OOP.

objectKarel helped students comprehend the technique and fundamental concepts of OOP (T1.6: Mean=2.96, Std. Dev.=1.13).
The analysis of students' replies showed that objectKarel provided slight or average support for 57% of the students, while for 37% of the students it provided a great deal of support. Overall, the vast majority of students (94%) reported that objectKarel helped them more or less in their transition from procedural to OOP and comprehension of OOP concepts. These results support the corresponding results of another study carried out in the same Department regarding the transfer of the knowledge acquired in objectKarel to Java afterwards (Xinogalos, 2012b):

- 34 out of 35 students stated that the introduction to the main OOP concepts with the microworld objectKarel helped them comprehend some of the concepts presented using Java afterwards; and
- the majority of students (65.7%—23 out of 35 students) stated that they did not have any difficulty in connecting the concepts taught in the context of the microworld with the corresponding concepts presented in Java.

General difficulties in learning programming
Students face several difficulties when learning programming. Some of these difficulties are specific to the programming technique and even the language used and other difficulties are independent of these factors and as such can be characterized as general difficulties.
With the aim of studying students' level of difficulty with such general difficulties a set of questions presented in Table 2 was included in the questionnaire. Some of these questions were adopted from the study carried out by Lahtinen et al. (2005) regarding the difficulties faced by novice programmers, while some more questions reflecting other important issues were added. Specifically:

- the issues with the codes T2.2, T2.3, T2.5, T2.11, T2.13 were based on the study by Lahtinen et al. (2005)
- the issue "Designing a program to solve a certain task" was substituted in our study by the following four issues: "Understanding the definition of a problem" (T2.7); "Selecting the appropriate structures (if, if/else, switch, for, while,...) for solving a problem" (T2.8); "Developing an algorithm for solving a problem" (T2.9) and " Transferring the algorithm to the programming language" (T2.10). These four issues represent the four distinct steps that we

encourage our students to follow even from the first lessons in order to solve a certain task. Emphasis is given on devising an algorithm, even conceptually, prior to start coding. The fact that it is cognitively complex to transfer the mental model or the description of an algorithm to a programming language has been recorded in the literature (Smith et al., 1994) and we wanted to investigate whether devising the algorithm in the first place is more or less difficult.

- "Finding bugs from my own program" (T2.13) was supplemented with "Understanding compilation error messages and correcting the corresponding errors" (T2.12), since syntax error messages are considered as well a source of difficulties for novices (Freund and Roberts, 1996).
- "Understanding programming structures" was split to "Understanding the role of the programming structures" (T2.4) in problem solving in general and "Learning the semantics of programming structures/concepts" (T2.6) that refers mainly to the way they work during program execution.

The most important results drawn from students' replies on the part of the questionnaire that referred to the difficulties faced while learning to program, independently of the technique, can be summarized as follows:

Difficulty 1 – Developing an algorithm

As we have already mentioned, we always encourage our students to start solving a programming task by first understanding the definition of a problem (T2.7: Mean=2.56, Std. dev.=1.07) and developing an algorithm for solving it (T2.9: Mean=2.68, Std. Dev=1.04), even conceptually. This process is quite difficult for nearly half the students. Specifically: 19% of the students reported that they face average difficulty and 27% much difficulty in understanding the definition of the problem in the first place; and 32% and 22% respectively in developing the algorithm.

Table 2. General difficulties of learning programming.

	Difficulty	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T2.1	Installing a programming environment	50	1.82	0.96	2	0.27	48	30	14	8	0
T2.2	Using programming environments	50	2.02	0.87	2	0.24	34	32	32	2	0
T2.3	Gaining access to computers/networks	50	1.34	0.69	1	0.19	76	16	6	2	0
T2.4	Understanding the role of programming structures	47	2.4	0.9	2	0.26	14	43	30	13	0
T2.5	Learning the programming language syntax	49	2.65	0.8	3	0.22	6	37	43	14	0
T2.6	Learning the semantics of programming structures/concepts	50	2.48	0.86	2	0.24	12	40	36	12	0
T2.7	Understanding the definition of a problem	48	2.56	1.07	2	0.3	17	37	19	27	0
T2.8	Selecting the appropriate structures (if, if/else, switch, for, while,...) for solving a problem	50	2.22	1.02	2	0.28	32	24	34	10	0
T2.9	Developing an algorithm for solving a problem (in paper and pencil, or conceptually)	50	2.68	1.04	3	0.29	14	30	32	22	2
T2.10	Transferring the algorithm to the programming language	50	2.8	1.03	3	0.29	10	30	34	22	4
T2.11	Dividing functionalities in functions (C) - classes (Java)	48	2.9	1.02	3	0.29	9	29	29	31	2
T2.12	Understanding compilation error messages and correcting the corresponding errors	50	2.9	1.02	3	0.28	10	26	34	24	6
T2.13	Finding bugs from my own program	49	2.88	1.11	3	0.31	12	23	39	18	8

Difficulty 2- Transferring an algorithm to a programming language

Transferring an algorithm to a programming language (T2.10: Mean=2.8, Std. Dev.=1.03) is considered by students even more difficult than developing the algorithm (T2.9: Mean=2.68, Std. Dev.=1.04). This might be attributed to:

- the *strictness* of programming languages and the underlying *notional machine* that students must learn to manipulate (du Boulay, 1989). Students do not comprehend easily that a program has to strictly comply with the rules of the system, and sometimes they are surprised by the level of detail required in programming mainly due to the *anthropomorphic characteristics* they attribute to the system.
- the fact that a step in an algorithm must be translated in a number of statements
- the fact that 57% of the students, as reported in the questionnaire, face average (43%) or much (14%) difficulty in mastering the syntax of the programming language (T2.5: Mean=2.65, Std. Dev.=0.8)
- the fact that 48% of the students, as reported in the questionnaire, face average (36%) or much (12%) difficulty with the semantics (T2.6: Mean=2.48, Std. Dev.=0.86) of the programming structures/concepts
- students' difficulties with programming structures that will be analyzed in the following paragraph.

Difficulty 3 – Programming structures

Familiarizing with programming structures is another source of difficulties for novice programmers. Based on students' replies, it is clear that understanding the role of programming structures in problem solving in general (T2.4: Mean=2.4, Std. Dev.=0.9), as well as learning the semantics of programming structures/concepts (T2.6: Mean=2.48, Std. Dev.=0.86) and selecting the appropriate ones for solving a problem is not easy (T2.8: Mean=2.22, Std. Dev.=1.02) for nearly half the students. Of course the difficulties with learning programming structures, as well as the notation of the programming language including both its syntax and semantics, are not new and were recorded back in 1986 by Sprohrer and Soloway and also by du Boulay in 1989. From the problems recorded by Sprohrer and Soloway (1986) as sources of difficulties in comprehending the semantics of programming structures the most prominent one based on our experience in teaching programming courses is the *human interpreter problem*: novices assume that the system will translate the structures in the same way they translate them.

Difficulty 4 – Modularization

Dividing the functionalities of a program in functions when using the procedural technique or classes when using the OOP technique (T2.11: Mean=2.9, Std. Dev.=1.02) is one of the most important difficulties for novices. Only 38% of the students reported having no or slight difficulty, while one third of students (33%) reported having great difficulty with dividing the functionalities of a program in functions/classes. Of course, experience has shown that students face difficulties not only with dividing the functionalities of a program in functions/classes, but even in implementing, combining and using them when they are specified by the teacher in the first place.

This is in accordance with the study by Sprohrer and Soloway (1986), which found that most of students' errors are due to difficulties in combining plans, or else parts of a program. An exhaustive list of the so called *plan composition problems* were recorded by Sprohrer and Soloway (1986) and are the following:

- *summarization problem*: complex combinations of plans in a main function overlooking potential dependencies in supporting functions.
- *optimization problem*
- *previous-experience problem or plan pollution problem*: existing plans are used incorrectly in new situations that seem relevant to students.
- *specification problem*: abstract plans developed by novices are not correctly adjusted in new situations.
- *natural-language problem*: plans in natural language are matched in the programming language used, leading to errors.
- *interpretation problem*: existing knowledge for goals and plans is used when interpreting the definition of a problem, ignoring requirements that are not explicitly mentioned or for which plans for fulfilling them cannot be easily recalled.
- *unexpected cases problem*: novices write programs that work for the most common situations and not all the potential situations.
- *boundary problem*
- *cognitive load problem*: results in overlooking small but important parts of plans or interactions between them.

These plan composition problems appear more or less in introductory programming courses and teachers should take care to device appropriate didactical situations for stressing such issues to students and supporting them in dealing with the corresponding problems they cause.

Difficulty 5 – Debugging

Debugging programs, along with modularization, is one of the most difficult issues in programming both in terms of understanding compilation error messages (T2.12: Mean=2.9, Std. Dev.=1.02) and finding bugs (T2.13: Mean=2.88, Std. Dev.=1.11), as reported by students. What seems surprising is that understanding compilation error messages that refer mainly

to syntax errors is equally difficult for students with finding and fixing logic errors. Are compilation messages so incomprehensible for novices? In some environments maybe, but in BlueJ error messages are quite user-friendly and moreover in objectKarel they are expressed in natural language without using any codes or advanced terminology. A possible explanation might be the result of an older study based on objectKarel. In the context of this study, students' successive compiled versions of several programs developed by them in groups of two were recorded using the *history of compilations* feature of objectKarel and analyzed, along with video recorded using Camtasia recorder (Xinogalos et al., 2006). This study showed that several students behave irrational when the system reports error messages and recompile their programs instantly without having made any change, or even make changes in random places in their program – a strong indication that for some time students do not even read carefully error messages!

In conclusion, *modularization* and *debugging* (difficulties 4 and 5) seem to be the greatest sources of difficulties for novice programmers, while *algorithm development* and *implementation* in a programming language are following (difficulties 1 and 2). What is interesting is that some students consider the implementation of an algorithm in a programming language more (even slightly) difficult than its development in the first place. This is due to difficulties with programming structures and not with the programming environments utilized that cause slight difficulties to students. The aforementioned difficulties (1, 2, 4 and 5) were recorded as the most difficult issues in programming with this or a slightly different form in the study by Lahtinen et al. (2005) as well. Specifically, Lahtinen et al. (2005) recorded the following issues: understanding how to design a program to solve a certain task, dividing functionality into procedures and finding bugs from their own programs. As Lahtinen et al. (2005, p. 16) state "*these are all issues where the student needs to understand larger entities of the program instead of just some details about it*".

Difficulties with imperative-procedural programming concepts/constructs

Extended research has been carried out on students' difficulties and misconceptions with imperative procedural programming concepts/constructs for several decades. Most of the studies were based on Basic and Pascal and recorded various difficulties and misconceptions for fundamental programming concepts. Pane and Myers (1996) have studied misconceptions that cause errors with *variables*, while Samurcay (1989) has categorized variables in different types and has recorded related difficulties. Variables are categorized based on distinct uses of the assignment operator, the functional meaning of variables and their processing in the context of loops. Sleeman et al. (1988) and Putman et al. (1989) have studied misconceptions and difficulties with *data input* and *assignment*, which of course relate to variables as well. Difficulties, conceptual models and misconceptions regarding flow of control for control structures (selection and repetition structures) have also been studied by several researchers (Hoc, 1989; Kahney, 1989; Kessler & Anderson, 1989; Putman et al., 1989; Rogalski and Samurcay, 1990; Sleeman et al., 1988). The results of the aforementioned studies were taken into account in devising educational material for the course. Students' difficulties are summarized in Table 3 and analyzed in the following paragraphs.

Table 3. Difficulties with imperative-procedural programming concepts.

	Programming concept/construct	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T3.1	Variables (life time, scope)	50	1.9	0.81	2	0.22	34	46	16	4	0
T3.2	Data types (int, float, char,...)	50	1.88	2	1.02	0.55	42	40	10	4	4
T3.3	Data input with Robert's libraries (x = GetInteger()...)	50	2.08	0.75	2	0.21	22	50	26	2	0
T3.4	Data input with C functions (scanf("%d", &x))	47	2.79	1.04	3	0.3	13	25	34	26	2
T3.5	Output	50	2	0.9	2	0.25	38	30	26	6	0
T3.6	Selection structures (if, if/else, switch)	50	2.1	1	2	0.28	36	28	26	10	0
T3.7	Repetition structures (for, while)	50	2.12	1.04	2	0.29	36	28	24	12	0
T3.8	Definition of functions	50	2.6	0.97	3	0.27	18	18	52	10	2
T3.9	Calling functions	50	2.56	1.01	3	0.28	16	32	34	16	2
T3.10	Parameters: declaring parameters	49	2.57	1.04	3	0.29	20	21	43	14	2
T3.11	Parameters: using the appropriate	50	2.66	1.04	3	0.29	18	20	42	18	2

	arguments										
T3.12	*Arrays*	50	3.06	1.24	3	0.34	12	24	22	30	12
T3.13	*Pointers*	50	3.5	0.97	4	0.27	2	16	24	46	12
T3.14	*Manipulating text files*	50	3.32	0.98	3.5	0.27	6	12	32	44	6
T3.15	*Using language libraries*	50	2.94	1.06	3	0.29	10	22	38	24	6

Pointers are one of the most difficult programming concepts, causing the majority of students much (46%) or very much (12%) difficulty, as recorded in their replies. This is in accordance with the study by Lahtinen et al. (2005). As a matter of fact, the two studies gave the same results (T3.13: Mean=3.5, Std. Dev.=0.97; Mean=3.59, Std. Dev.=1.04 in (Lahtinen et al., 2005)). Pointers are an abstract concept without corresponding examples from students' everyday life and as such cognitively complex to understand (Lahtinen et al., 2005). Students reported facing difficulties in manipulating text files as well (T3.14: Mean=3.32, Std. Dev.=0.98). However, this was expected since manipulation of text files requires the usage of pointers and moreover language libraries.

Arrays are reported as the next more difficult programming concept for students (T3.12: Mean=3.06, Std. Dev.=1.24). Data structures are well known to cause difficulties to students. Although arrays are one of the simplest data structures they are still not easy for students to use in the context of a program. Examples from everyday life exist and students seem to comprehend the concept of arrays, but they have problems in implementing programs with arrays. Based on several years of experience on assessing students' assignments and exam solutions with arrays the most important difficulties related to arrays seem to be the following: using the correct indexes for traversing arrays, especially two dimensional; making calculations per row and even worst per column (for example, mean per row); using arrays as parameters and so on.

Using *language libraries* is reported as another common source of difficulties for novices (T3.15: Mean=2.94, Std. Dev.=1.06). Searching in language libraries, finding the appropriate function and using it correctly are difficult for novices. The idea to use specially designed libraries for novices that hide the complexity of language constructs and library functions seems to be a good idea. A characteristic example is the use of Professor's Eric Robert (Stanford University) *GetType()* functions described in his textbook "The Art and Science of C" for data input instead of the *scanf("format", &variablename)* C function. Students reported that they find more difficult the use of the *scanf* (T3.4: Mean=2.79, Std. Dev.=1.04) instead of the *GetType* function (T3.3: Mean=2.08, Std. Dev.=0.75). More than one fourth of students (28%) report that they have much of very much difficulty on using the *scanf* function in contrast with just 2% with the *GetType* functions that do not make reference to memory addresses, while format is more easily defined by the *Type* part of the function.

As recorded in the previous section, dividing the functionalities of a program in *functions* when using the procedural technique (T2.11: Mean=2.9, Std. Dev.=1.02) was reported as one of the most important difficulties for novices. However, students face difficulties even when the decomposition of a problem is provided and they have to define a subprogram for each defined sub-problem. Specifically, more than half the students reported that they face difficulties with defining functions (T3.8: 64%), declaring their parameters (T3.10: 59%) and calling functions (T3.9: 52%) using the appropriate arguments (T3.11: 62%).

Finally, the majority of students state that face less difficulties – in descending order – with *repetition structures*, *selection structures*, *output*, *data types* and *variables*. These programming concepts/constructs have been presented in detail to the majority of our students during their last year studies in Secondary education and this is considered to be the most important reason for dealing with them more easily. However, regarding variables it must be stressed out that students most probably answered having in mind declaration of variables and overlooked the life time and scope aspects. If they were asked more explicitly to express their difficulty with variables' scope and life time most probably different results would be recorded.

Generally speaking the aforementioned results are in compliance with the author's experience on teaching the Introductory Computer Programming course at a Technology Management Department for several years. Based on this experience the schedule and course outline presented in Table 4 has been adopted the last years, allocating more time to concepts/constructs that are more difficult for students to comprehend.

Table 4. The contents of the Computer Programming course.

Week	Learning Unit	Programming concepts/constructs
1	Introduction	programming languages and environments development, compilation, debugging and execution of a program
2	Sequential structure	variables (life time, scope), data types , data input with Robert's libraries, assignment, expressions, output
3	Selection structures	if, if/else, if/else if, switch, nested selection structures
4	Repetition structures	for, while, nested selection and repetition structures
5, 6	Functions	abstraction and modularization, defining and calling functions and procedures, parameters and arguments

7	Arrays	one and two dimensional arrays, basic operations with arrays
8	Arrays and Functions	arrays as parameters
9, 10	Pointers	addresses as data, pointer operators, call by reference with the use of pointers
11	Characters and strings	manipulation of characters with the ctype.h library, strings as arrays of characters and pointers, manipulation of strings with the string.h library
12, 13	Text files	character or line manipulation of text files, formatted input/output from/to text files

Difficulties with OOP concepts/constructs

Several studies have been carried out with the aim of recording students' difficulties and misconceptions (Holland et al. 1997; Carter and Fowler 1998; Ragonis and Ben-Ari 2005; Garner et al. 2005; Thomasson et al. 2006; Sanders and Thomas 2007; Sanders et al. 2008), and less often students' conceptions (Eckerdal and Thuné 2005; Teif and Hazzan 2006) when introduced to OOP. Common difficulties and misconceptions recorded in the literature include the following: *object/class conflation* (Holland et al. 1997; Garner et al. 2005; Sanders et al. 2008; Sanders and Thomas 2007); *object/variable conflation* (Carter and Fowler 1998); *object/record conflation* (Holland et al. 1997); *class/collection of objects conflation* and various variations (Ragonis and Ben-Ari 2005; Thomasson et al. 2006; Sanders and Thomas 2007; Teif and Hazzan 2006); *class-object as set-subset* or *class-object as whole-part* misconception (Teif and Hazzan 2006); *difficulties with modeling* (Thomasson et al. 2006; Sanders and Thomas 2007; Sanders et al. 2008; Eckerdal and Thuné 2005); distinction between *static and dynamic aspects of OO* (Sanders et al. 2008; Ragonis and Ben-Ari 2005); difficulties with *abstraction techniques* including inheritance, polymorphism, overriding, abstract methods and interfaces (Garner et al., 2005; Or-Bach and Lavy, 2004); difficulties with *accessor and mutator methods* (Garner et al., 2005); and *constructors* (Fleury, 2000). As was the case with the first programming course, the aforementioned studies (and many more) were taken into account in devising educational material for the second course on object-oriented design and programming. Students' difficulties are summarized in Table 5 and analyzed in the following paragraphs.

Table 5. Difficulties with OOP concepts.

	Programming concept/construct	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T5.1	Object	50	1.9	0.84	2	0.23	36	42	18	4	0
T5.2	Class	50	1.96	0.92	2	0.26	38	34	22	6	0
T5.3	Fields (role, scope, access)	50	2.06	1.08	2	0.3	38	30	24	4	4
T5.4	Constructor	50	2	1.12	2	0.31	46	20	26	4	4
T5.5	Methods	49	2.16	1.05	2	0.29	33	28	33	2	4
T5.6	Accessor methods (*getters*)	50	1.96	1.09	2	0.3	46	22	26	2	4
T5.7	Mutator methods (*setters*)	49	1.98	1.09	2	0.31	45	22	27	2	4
T5.8	Access modifiers (public, private, protected)	48	2.15	1.07	2	0.3	31	38	21	6	4
T5.9	*Object interaction (i.e. an object containing other objects)*	50	2.66	1.12	2	0.31	12	40	26	14	8
T5.10	*Internal method call*	50	2.8	1.07	3	0.3	10	32	32	20	6
T5.11	*External method call (dot notation)*	50	2.9	1.07	3	0.3	10	26	34	24	6
T5.12	*Object collections (i.e ArrayList): comprehending its functionality*	50	3.32	1.19	4	0.33	8	20	18	40	14
T5.13	*Object collections (i.e ArrayList): writing code for manipulating it*	50	3.38	1.16	4	0.32	6	20	20	38	16
T5.14	Using class libraries	50	3.02	1.06	3	0.29	6	26	38	20	10
T5.15	Inheritance: extending an existing class	47	2.85	1.4	3	0.4	24	19	21	21	15
T5.16	*Inheritance: refactoring a project for improving its structure with inheritance*	49	3.08	1.24	3	0.35	12	20	29	25	14

T5.17	*Polymorphism*	49	3.45	1.12	4	0.31	6	12	31	33	18
T5.18	*Overriding*	46	3.67	1.06	4	0.31	4	7	30	35	24
T5.19	*Abstract classes*	49	3.53	1.04	4	0.29	6	6	33	39	16
T5.20	*Interfaces*	49	3.57	1.02	4	0.29	4	10	27	43	16

Abstraction techniques in OOP are considered to be one of its most important strengths. However, as both teachers' experience and students' replies in the questionnaire show, abstraction techniques cause great difficulties in students. Students reported that they face the following difficulties in ascending order:

- 36% of the students face much or very much difficulty in using *inheritance* for extending an existing user defined or library class (T5.15: Mean=2.85, Std. Dev.=1.4).
- even more students (39%) face much or very much difficulty in *refactoring* a project for improving its structure using inheritance (T5.16: Mean=3.08, Std. Dev.=1.24).
- half the students (51%) face much or very much difficulty in using *polymorphism* for further improving the structure of a project using inheritance (T5.17: Mean=3.45, Std. Dev.=1.12).
- more than half the students (55%) face much or very much difficulty in using *abstract classes* (T5.19: Mean=3.53, Std. Dev.=1.04). Students cannot easily comprehend the true meaning of abstract classes and they tend to declare a superclass as an abstract class only in cases where they have to declare an abstract method in order for their project to compile.
- the majority of students (59%) face much or very much difficulty with *interfaces* (T5.20: Mean=3.57, Std. Dev.=1.02). Interfaces seem to be a very abstract concept for students to comprehend. Students do not easily understand what the value of a class that contains just declaration of abstract methods and no implementation is.
- the majority of students (59%) face much or very much difficulty in overriding inherited methods (T5.18: Mean=3.67, Std. Dev.=1.06).

Object collections, such as ArrayLists, were reported as the next more difficult concept for the majority of students (54%), both in terms of comprehending their functionality (T5.12: Mean=3.32, Std. Dev.=1.19) and writing code for manipulating them (T5.13: Mean=3.38, Std. Dev.=1.16). However, this was more or less expected due to the wide known difficulties with data structures in general, as well as students' difficulties with using *class libraries* that was recorded in this study also (T5.14: Mean=3.02, Std. Dev.=1.06). As a matter of fact, a study that was carried out in the context of the same course showed that many difficulties are not related to ArrayLists in particular, but in key OOP concepts that students must have mastered prior to their exposure to object collections (Xinogalos, 2010). More specifically, the following key concepts have to be deeply comprehended prior to the presentation of ArrayLists: object types and not just primitive types can be used as local variables, parameters and return types; classes can have fields/attributes of some object type (collaborative classes); access modifiers; accessor methods and especially their actual usefulness for accessing private fields outside their class; internal and external method calls.

Some of the aforementioned key concepts that are important for comprehending and manipulating ArrayList object collections are also difficult for students. The survey results showed that *interacting objects* (T5.9: Mean=2.66, Std. Dev.=1.12), *internal* (T5.10: Mean=2.8, Std. Dev.=1.07) and *external method calls* (T5.11: Mean=2.9, Std. Dev.=1.07) are the concepts that follow ArrayLists in terms of difficulty. *Methods* (T5.5: Mean=2.16, Std. Dev.=1.05), *access modifiers* (T5.8: Mean=2.15, Std. Dev.=1.07) and *fields* (T5.3: Mean=2.08, Std. Dev.=1.08) also cause average difficulty in several students.

Generally speaking the aforementioned results are in compliance with the author's experience on teaching the Object-Oriented Design and Programming course at a Technology Management Department for several years. Based on this experience the schedule and course outline presented in Table 6 has been adopted the last years.

Table 6. The contents of the Object Oriented Design and Programming course.

Week	Learning Unit	Programming concepts/constructs
1	Basic concepts in objectKarel	objects, classes and inheritance in objectKarel
2	Advanced concepts in objectKarel	multilevel inheritance, polymorphism and overriding
3	Class definition	fields, constructors, accessor and mutator methods, return statements, parameters (formal, actual), variable scope/lifetime, conditional statements
4	Static methods	instance vs. static/class methods, the main method, executing without BlueJ, byte code, Java Virtual Machine
5	Object interaction	abstraction, modularization, objects creating objects, multiple constructors (overloading), class diagram, object diagram, primitive and object types,

		internal/external method call
6, 7	Objects collections	flexible size collections (ArrayList), fixed size collections (array), generic classes, iterators, loops (while, for, for-each)
8	Class libraries	Java standard class library, reading documentation, interface vs. implementation of a class, exploring and using classes (Scanner, Random, HashMap, HashSet)
9	Inheritance	inheritance, superclass, subclass, inheritance hierarchy, superclass constructor, subtyping, substitution, autoboxing
10	Polymorphism	Polymorphism, overriding: static/dynamic type, dynamic method lookup, super (in methods), protected access
11	Abstract classes and interfaces	abstract classes and interfaces
12	Simple application	analysis, design and implementation of a simple application
13	Introduction in GUI programming	main elements of a GUI in Java

Teaching programming

The overall teaching and learning design for both courses is based on the same rationale that is summarized in the following paragraphs, while students' assessment for the support provided by each aspect of this learning design is presented in Table 7:

- The concepts presented in the course are *organized in clear learning objects*, or else units. The material is organized in a series of lectures and corresponding labs, with each week of teaching having specific and transparent didactical aims acknowledged to students at the beginning of the semester. Students can find all the didactical material used – slides, programming leaflets, programs and so on – organized in different folders that can be accessed through the University's Course Management System. Students assess this organization as much (60%) or very much (32%) important (T7.1: Mean=4.2, Std. Dev.=0.73).
- The last few years an attempt to use practical activities sheets during lectures and decreasing theoretical presentations has been made, in order to engage students and deal with the low attendance rates in lectures (optional), in comparison with the very high attendance rates in lab sessions (compulsory). *Active-learning* is currently attracting instructors' and researchers' interest and is highly appreciated by the students, as recorded in this study (T7.2: Mean=4.28, Std. Dev.=0.83). However, we must mention that preparing such material is quite challenging, as active-learning activities are time consuming and if they are not carefully designed they might not attract students' attention or fail to communicate the intended concepts. Moreover, in the introductory programming course there is much pressure in terms of the time available for familiarizing novices and dealing with their difficulties regarding various aspects of programming (du Boulay, 1989): orientation – what is programming all about; the notional machine and its relation with the physical machine; notation; structures; and pragmatics.
- Hands-on activities are considered necessary for acquiring problem solving skills and programming capabilities. For this reason, both courses include *programming exercises* carried out at *labs* and as *homework*. Students also realize this fact and consider solving exercises at labs and as homework as important. Specifically, nearly all the students (98%) consider solving exercises at labs as much or very much important (T7.3: Mean=4.68, Std. Dev.=0.59), constituting this aspect of the course as the most important of all. Weekly homework is considered as much or very much important by 80% of the students (T7.4: Mean=4.18, Std. Dev.=0.8), while announcing the solutions instantly after the submission deadline of the assignments is perceived as even more important by students (T7.5: Mean=4.4, Std. Dev.=0.95).
- Another important aspect of the learning design is the *mid-semester exams* that count - as is the case for homework - for the final grade. Mid-semester exams keep students vigilant and give instructors the ability to check students' achievements and make appropriate interventions for dealing with their difficulties before it is too late. As is the case with homework, although students have to work a lot during the semester so as to be prepared for the exams they realize that this is necessary for such a cognitively demanding discipline. Four out of five students believe that writing mid-semester exams provides much (36%) or very much (44%) support in learning programming (T7.6: Mean=4.2, Std. Dev.=0.86).

Table 7. Support provided by various teaching aspects.

	Support provided by …	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T7.1	Organizing the material in a series of lectures and corresponding labs	50	4.2	0.73	4	0.20	2	0	6	60	32
T7.2	Using practical activities sheets during lectures and decreasing theoretical presentations	50	4.28	0.83	4	0.23	2	2	6	46	44
T7.3	Solving exercises at labs	50	4.68	0.59	5	0.16	0	2	0	26	72
T7.4	Assigning weekly homework for each lecture/lab	50	4.18	0.8	4	0.22	0	2	18	40	40
T7.5	Announcing the solutions of weekly assignments instantly after the submission deadline	50	4.4	0.95	5	0.26	2	4	8	24	62
T7.6	Writing mid-semester exams	50	4.2	0.86	4	0.24	0	4	16	36	44

An important role in realizing the aforementioned learning design is played by the University's Learning Management System (LMS) or to be more precise Course Management System. This system is used for enhancing the learning experience of students by providing them access to the educational material that is organized in lectures (T7.1, T7.2), assigning weekly assignments (T7.4) and announcing their solutions (T7.5).

Learning programming

In this section, students' perceptions regarding the support provided in learning programming by various alternatives that can be used for applying the aforementioned teaching and learning design are analyzed. As can be seen in Table 8:

- Students believe that they learn to program mostly during *practicing programming*. The majority of students believe that learn programming mostly during writing programs either personally (T8.3: Mean=3.96, Std. Dev.=0.9) or collaboratively (T8.4: Mean=4.06, Std. Dev.=1.02) in lab sessions, and carrying out programming assignments as homework either personally (T8.6: Mean=3.82, Std. Dev.=1.1) or with classmates (T8.7: Mean=3.82, Std. Dev.=1). An important aspect that must be stressed out is the fact that several students find it motivating writing programs collaboratively with classmates. This is applied in labs, but it is offered as a choice for homework as well, as long as students inform the instructor for the teams formed. Of course, technological solutions for distributed pair programming are nowadays available at least for Java programming and their incorporation is believed to deliver several advantages, such as more balanced contribution to the development of a program and ability to check students' participation (Tsompanoudi et al., 2013).
- Lectures seem to motivate students (T8.1: Mean=3.26, Std. Dev.=0.85) less than labs and this explains the smaller attendance rate in lectures in comparison to labs, which however are compulsory. However, carrying out *activities in lectures* is considered by students to support them in learning programming (T8.2: Mean=3.63, Std. Dev.=0.86). The majority of students (67%) state that much or very much help is provided by carrying out activities during lectures and this clearly indicates that special attention should be paid by instructors in preparing relevant material and making lectures more engaging for students.

Table 8. Students' perceptions regarding various ways of learning programming.

	I learn programming during …	N	Mean	Standard Deviation	Median	Confidence	Not at all (%)	Slightly (%)	Averagely (%)	Much (%)	Very much (%)
T8.1	Lectures	50	3.26	0.85	3	0.24	2	14	46	32	6
T8.2	Carrying out activities in lectures	49	3.63	0.86	4	0.24	0	14	19	57	10
T8.3	Problem solving personally in labs	50	3.96	0.9	4	0.25	0	6	24	38	32
T8.4	Problem solving collaboratively in labs	50	4.06	1.02	4	0.28	0	10	18	28	44

T8.5	Studying alone	50	3.5	1.15	4	0.32	4	18	24	32	22
T8.6	Carrying out programming assignments alone	50	3.82	1.1	4	0.3	2	12	22	30	34
T8.7	Carrying out programming assignments with classmates	50	3.82	1	4	0.28	0	12	24	34	30

Material for Study

Students learn programming mainly in the context of programming exercises carried out either in labs or as homework, as recorded in their replies to the relevant questions analyzed in the previous paragraphs. The material used for such exercises and assignments (T9.4: Mean=4.06, Std. Dev.=0.82) together with the exemplary solutions provided (T9.5: Mean=4.22, Std. Dev.=0.82) is also highly appreciated by the majority of students (78% and 84% respectively) as the main material for studying. Lab and homework programming exercises are considered to provide more support than other programs provided to students for studying (T9.3: Mean=3.16, Std. Dev.=1.13), probably because of their involvement, time and effort devoted in developing them. When it comes to the necessary material used for studying the theoretical aspects of programming concepts, students seem to prefer mostly the presentations prepared by the instructor and used in lectures (T9.2: Mean=3.8, Std. Dev.=0.88) and less the course textbooks (T9.1: Mean=3.02, Std. Dev.=1.15) no matter how good they are. These results clearly indicate that instructors have to invest a lot of effort and time for preparing high quality material both for theoretical and practical aspects of a programming course.

Table 9. Support provided to students by different kinds of material for studying.

	Support provided by …	**N**	**Mean**	**Standard Deviation**	**Median**	**Confidence**	**Not at all (%)**	**Slightly (%)**	**Averagely (%)**	**Much (%)**	**Very much (%)**
T9.1	Course textbook	50	3.02	1.15	3	0.32	6	32	30	18	14
T9.2	Lecture presentations	50	3.8	0.88	4	0.24	2	2	32	42	22
T9.3	Programs	50	3.16	1.13	3	0.31	8	16	32	32	12
T9.4	Exercises/assignments	50	4.06	0.82	4	0.23	0	4	18	46	32
T9.5	Exemplary solutions of weekly assignments	50	4.22	0.82	4	0.23	0	4	12	42	42
T9.6	Material I found in the web	48	2.04	1.07	2	0.3	38	35	15	10	2

CONCLUSIONS

Designing and deploying programming courses is undoubtedly a challenging task. In this paper, an attempt to analyze important aspects of a sequence of two courses on imperative-procedural and object-oriented programming in a non-CS majors Department is made. This analysis is based on a questionnaire filled in by fifty students in a voluntary basis. The issues of the programming courses that are analyzed refer to: the *strategy* selected for the introduction to programming; the sequence of the *programming techniques and languages* taught and the transition from the one to the other; students' *difficulties* with programming in general and with imperative-procedural and object-oriented programming in specific; the *teaching* and *learning design* of both courses; and the *material* used for learning programming. Based on the analysis of the questionnaire's results the following conclusions can be drawn.

First of all, it seems that *the introduction to programming using a pseudo-language is a good choice*. Pseudo-languages are less strict than conventional programming languages and help students concentrate on the most important aspects, which are without doubt the algorithmic/programming concepts/constructs and not their syntax. Students that had been introduced to programming with a pseudo-language, during their last year of studies in Secondary Education, stated that they were significantly supported in understanding the programming concepts/constructs when taught imperative programming at University with C. Nowadays, several environments are freely available for supporting an introduction to programming using pseudo-languages, or even flowcharts. A comparative analysis of *flowchart-based programming environments* that incorporate structure editors for developing algorithms easily, program animation features for running flowcharts in a step by step manner and abilities of automatically generating the corresponding source code is provided in (Xinogalos, 2013).

The transition from imperative-procedural to object-oriented programming is not easy for students. As a matter of fact the results of the questionnaire showed that this transition is considered by students even more difficult than their introduction to programming. Experience on teaching both courses has shown that students find it difficult to change their mindset and utilize classes and objects instead of functions as the main concepts for devising a solution to a problem. The author's and instructor's

belief is that *students have to be supported both during their introduction to programming and their transition from one programming technique to another*, no matter which technique is used first. In our case, students are supported on their introduction to programming with the imperative language C mainly by using Prof. Roberts' libraries that hide some subtleties of the C programming language (Roberts, 1994). Students are also supported by adopting an algorithmic way of presenting programming concepts (i.e. using a pseudo-language) and solving programming tasks, prior to presenting them in C. For the transition to object-oriented programming we use the programming microworld objectKarel for a hands-on, playful and clear presentation of fundamental OOP concepts prior to using Java and the educational programming environment BlueJ. Based on the questionnaire results it is apparent that students evaluate positively the sequence of techniques and languages taught and the measures taken for supporting them in their transition from the one technique to the other.

Learning programming is accompanied with several intrinsic difficulties, which cannot be dealt with easily. The sequence of courses presented in this paper was designed taking into account the research carried out the last decades regarding the teaching and learning of programming and the accompanying difficulties. So, the aim of this study was definitely not to investigate what students' difficulties with programming are, but to investigate what difficulties insist to appear and whether the whole design of the courses and the learning units were appropriately distributed. Based on the questionnaire results, the most important students' general difficulties not related to a specific programming technique are: developing an *algorithm* for solving a problem, dividing its functionalities in *functions/classes*, transferring it to the *programming language* and *debugging* it both in terms of syntax and logical errors. A course on algorithms, which is missing from the program of studies, would definitely help students in acquiring problem solving capabilities. *Pointers*, *arrays*, *language libraries* and *functions* are the most challenging concepts for students in the context of the first imperative-procedural programming course, while in the object-oriented course the most challenging concepts refer to abstraction techniques – *inheritance*, *polymorphism*, *overriding*, *abstract classes* and *interfaces* – and *object collections*. Knowing students' difficulties is important for the instructor in order to distribute appropriately the available time to the various learning units and devise special didactical situations for supporting students in dealing with the underlying difficulties.

Moreover, special attention has to be paid in the overall teaching and learning design of a programming course. As indicated by the results of the questionnaire, as well as the instructor's experience, it is important for students to:

- *Organize the material in clear learning objects*, taught in a series of lectures and corresponding labs with each week of teaching having specific and transparent didactical aims acknowledged in advance to students.
- *Utilize active-learning activities in lectures* for attracting students' attention and dealing with low attendance rates in lectures, in contrast with labs that have a more hands-on nature.
- *Carry out programming exercises* at *labs*, since such exercises are considered by students as a very important aspect of a programming course. Programming exercises assigned as *homework*, although time-consuming and demanding are also considered important. Students believe that they *learn to program mostly during practicing programming* either personally or collaboratively. In the later case, the advantages provided by *distributed pair programming* should be considered by instructors. Also, a technological solution for an *automatic assessment* of students' programs is important both for providing immediate feedback to students and saving time for instructors.
- *Carry out mid-semester exams* for monitoring students' progress and making appropriate interventions for dealing with their difficulties in time. Both homework and mid-semester exams count for the final grade.
- *Use an LMS for organizing and managing the course, keeping students informed for the learning design, providing easy access to the educational material from anywhere, enhancing communication and collaboration and providing guidance to students.* An LMS is of vital importance for applying the aforementioned teaching and learning design, engaging students' and instructors in the whole process and providing capabilities for blended learning.

Finally, when it comes to the material used by students for studying it is clear that various resources are and should be available for students. However, *students rely heavily on the material prepared by instructors and used in lectures and labs for studying*. Consequently, it is very important for instructors to prepare high quality presentations, lab exercises, assignments and exemplary solutions in order to support students in learning programming.

The conclusions drawn from the study presented in this paper can be used as guidelines for improving the quality and effectiveness of existing programming courses, as well as for designing and deploying new programming courses. A limitation of the study is certainly the fact that it was based on a sample of fifty students and so its conclusions need to be further confirmed. However, another important fact is that these results comply with the instructor's experience on teaching and reforming the two programming courses for several years. This study was completed by chance during a time when the Department of Technology Management (the Department the study took place) was merged with a Department of Applied Informatics and the establishment of a new School of Information Sciences, with three times more students entering the new Department each academic year. Hopefully, this will give us the chance to investigate further the issue and validate the results of this study in a different context.

REFERENCES

Bennedsen, J. & Caspersen, M. (2004), Programming in Context – A Model-First Approach to CS1, *Proceedings of SIGCSE '04,* 477-481.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller P. (1997). Mini-languages: A way to learn programming principles. *Education and Information Technologies*, 2, 65-83.

Carter, J. & Fowler, A. (1998). Object oriented students? *SIGCSE Bull.* 30, 3, 271.

Cooper, S., Dann, W. and Pausch, R. (2003). Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (SIGCSE '03). ACM, New York, NY, USA, 191-195.

Decker, R. & Hirshfield, S. (1994), The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught In CS1, ACM SIGCSE Bulletin, Vol. 26, No. 1, 51-55.

Du Boulay, B. 1989. Some Difficulties of Learning to Program, *Studying The Novice Programmer*. E. Soloway & J. Sprohrer (Eds.), Lawrence Erlbaum Associates, 283-300.

Eckerdal, A. & Thuné, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05)*. ACM, New York, NY, USA, 89-93.

Fleury, A. (2000). Programming in Java: student-constructed rules. *ACM SIGCSE Bulletin*, Vol. 32, Issue 1, 197-201.

Freund, S. N. & Roberts, E. S. 1996. THETIS: An ANSI C programming environment designed for introductory use. ACM SIGSCE Bulletin, Vol. 28, No. 1, 300-304.

Garner, S., Haden, P. and Robins, A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proc. of the 7th Australasian conference on Computing education - Volume 42 (ACE '05)*, Alison Young and Denise Tolhurst (Eds.), Vol. 42. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 173-180.

Georgantaki, S. & Retalis, S. (2007). Using Educational Tools for Teaching Object Oriented Design and Programming, *Journal of Information Technology Impact*, Vol. 7, No. 2, pp. 111-130.

Hadjerrouit, S. (1998), A Constructivist Framework for Integrating the Java Paradigm into the Undergraduate Curriculum, *ACM SIGCSE Bulletin, Vol. 30, Issue 3*, 105-107.

Hadjerrouit, S. (1999), A constructivist approach to object-oriented design and programming, *ACM SIGCSE Bulletin, Vol. 31, Issue 3*, 171-174.

Hoc, J. (1989) Do We Really Have Conditionals In Our Brains? In *Studying The Novice Programmer*, Soloway, E., Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, 179-190.

Holland, S. Griffiths, R., Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin*, Vol. 29, No. 1, 131-134.

Kahney, H. (1989) What Do Novice Programmers Know About Recursion? In *Studying The Novice Programmer*, Soloway, E., Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, pp. 209-228.

Kaplan, R. M. 2010. Choosing a first programming language. In *Proceedings of the 2010 ACM conference on Information technology education* (SIGITE '10). ACM, New York, NY, USA, 163-164. DOI=10.1145/1867651.1867697 http://doi.acm.org/10.1145/1867651.1867697

Kessler, C. & Anderson, J. (1989) Learning Flow of Control: Recursive and Iterative Procedures. In *Studying The Novice Programmer*, Soloway, E., Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, pp. 229-260.

Lahtinen, E., Ala-Mutka, K. & Jarvinen, H. 2005. A Study of Difficulties of Novice Programmers. In: *Innovation and Technology in Computer Science Education 2005*, 14–18.

McIver, L. and Conway, D. 1996. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)* (SEEP '96). IEEE Computer Society, Washington, DC, USA, 309-.

Or-Bach, R. and Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin*, 36(2), 82–86.

Pane, J. F. and Myers, B.A. (1996) Usability Issues in the Design of Novice Programming Systems, Technical Report CMU-CS-96-132, School of Computer Science, Carnegie Mellon University (also available as: Human-Computer Interaction Institute Technical Report CMU-HCII-96-101).

Parker, K., J. T. Chao, Ottawa, T. and Chang, J. 2006. A Formal Language Selection Process for Introductory Programming Courses. *Journal of Information Technology Education*, Vol. 5, 133-151.

Putman, R., Sleeman, D., Baxter, J. & Kuspa, L. (1989) A Summary Of Misconceptions Of High-School BASIC Programmers. In *Studying The Novice Programmer*, Soloway, E., Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, pp. 301-314.

Ragonis, N. & Ben-Ari, M. (2005). A Long-Term Investigation of the Comprehension of OOP Concepts by Novices. *International Journal of Computer Science Education*, 15(3), 203-221.

Roberts, Eric. (1994). The Art and Science of C: A Library Based Introduction to Computer Science. Prentice Hall.

Robins, A., Rountree, J & Rountree N. (2003). Learning and Teaching Programming: A review and Discussion. *Computer Science Education*, Vol. 13, Issue 2, 137-172.

Rogalski, J. & Samurcay, R. (1990) Acquisition of Programming Knowledge and Skills. *In Psychology of Programming*, Hoc, J., Green, T., Samurcay, R. and Gilmore, D. (Eds.), Academic Press, pp. 157-174.

Samurcay, R. (1989) The Concept of Variable in Programming: its Meaning and Use in Problem-Solving by Novice Programmers. *In Studying The Novice Programmer*, Soloway, E. and Sprohrer, J. (Eds.), Lawrence Erlbaum Associates, pp. 161-178.

Sanders, K. & Thomas, L. (2007). Checklists for grading object-oriented CS1 programs: concepts and misconceptions. In *Proc. of the 12th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '07)*. ACM, New York, NY, USA, 166-170.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J., Thomas, L. & Zander, C. (2008). Student understanding of object-oriented programming as expressed in concept maps. In *Proc. 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)*. ACM, New York, NY, USA, 332-336.

Sleeman, D., Putman, R., Baxter, J. & Kuspa, L. (1988) An Introductory Pascal Class: A Case Study Of Students' Errors. *In Teaching and Learning Computer Programming*, Mayer, R. (Ed.), Lawrence Erlbaum Associates, pp. 237-258.

Smith, D.C., Cypher, A. & Sprohrer, J. 1994. KIDSIM: Programming Agents Without a Programming Language. *Communications of the ACM*, Vol.37, No.7, 55-67.

Spohrer, J. C. & Soloway, E. (1986) Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM, Vol. 29, No. 7*, pp. 624-632.

Teif, M. & Hazzan, O. (2006). Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts. In *Working group reports on ITiCSE on Innovation and technology in computer science education (ITiCSE-WGR '06)*. ACM, New York, NY, USA, 55-60.

Tempte, M C. (1991), Let's Begin Introducing the Object-Oriented Paradigm, *ACM SIGCSE Bulletin, Vol. 23, No. I*, 338-342.

Thomasson, B., Ratcliffe, M. & Thomas, L. (2006). Identifying novice difficulties in object oriented design. *SIGCSE Bull.* 38, 3 (June 2006), 28-32.

Tsompanoudi, D., Satratzemi, M. and Xinogalos, S. (2013). Exploring the effects of Collaboration Scripts embedded in a Distributed Pair Programming System. *Proceedings of the 18th ACM ITiCSE Conference*, 1-3 July 2013, Canterbury UK, 225-230.

Wick, M. (1995), On Using C++ and Object-Orientation in CS1: the Message is still more important than the Medium, *ACM SIGCSE Bulletin, Vol. 27, Issue 1*, 322-326.

Xinogalos, S. (2010). Difficulties with Collection Classes in Java – The Case of the ArrayList Collection. *Proceedings of the 2nd International Conference on Computer Supported Education (CSEDU)*, 7-10 April, Valencia, Spain, 120-125.

Xinogalos, S. (2012a). Programming Techniques and Environments in a Technology Management Department. *Proceedings of the 5th Balkan Conference in Informatics (BCI 2012)*, 16-20 September, Novi Sad, Serbia, ACM, New York, NY, USA, 136-141.

Xinogalos, S. (2012b). An Evaluation of Knowledge Transfer from Microworld Programming to Conventional Programming. *Journal of Educational Computing Research*, Vol. 47, Number 3/2012, 251-277.

Xinogalos, S. (2013). Using Flowchart-based Programming Environments for Simplifying Programming and Software Engineering Processes. *In Proceedings of 4th IEEE EDUCON Conference*, Berlin, Germany, 13-15 March 2013, IEEE Press, 1313-1322.

Xinogalos, S., Satratzemi, M. & Dagdilelis, V. (2006). An Introduction to object-oriented programming with a didactic microworld: objectKarel. *Computers & Education*, Volume 47, Issue 2, September 2006, 148-171, Elsevier Publishers.