
PROBLEMS IN LEARNING AND TEACHING PROGRAMMING

- a literature study for developing visualizations in the Codewitz-Minerva project

Kirsti Ala-Mutka, Institute of Software Systems, Tampere University of Technology, Finland

Email: kirsti.ala-mutka@tut.fi

1. Introduction

The art of programming includes knowledge of programming tools and languages, problem-solving skills, and effective strategies for program design and implementation. A common approach in programming education is to first teach the basics of a programming language and then guide students towards effective strategies for the whole programming process. Therefore, the learning of the basic concepts is often emphasized; these form the basis for building more advanced skills. Codewitz project [5] aims at fostering the learning of the basic concepts and structures by creating dynamic visual learning materials for students. In order to create effective visualizations and educational strategies for their use, it is necessary to review the research previously published in the field.

This literature study aims at a brief overview on the problems and on visualization solutions in programming education. The emphasis is on novice programmers that are beginning their programming studies; these courses are often referred in the literature as CS1 courses. An excellent source for the study has been a recent review on the research relating to the educational study of programming, written by Robins et. al. [27]. Another vast source of information is an older collection of research papers on novice programmers, edited by Soloway and Spohrer [29]. Also the research by Milne and Rowe [21], who studied students' and tutors' perceptions of difficulties in object-oriented programming, relates closely to our work. However, their work differs from our needs analysis survey in that the group of respondents was considerably smaller (altogether 66 students and teachers) and the questions concentrated on the C++ language concepts. In addition, many other articles contain important information for the Codewitz-Minerva project and will be mentioned later.

The structure of this study is organized as follows. First, Section 2 recognizes some characteristics and problems of novice programmers. In Section 3, we discuss educational visualization approaches for programming. Section 4 concludes the study and proposes some

approaches for the project. Finally, Section 5 lists the references that have been recognized as important resources for the project participants and other interested readers.

2. Difficulties in learning programming

Learning to program is generally considered hard, and programming courses often have high dropout rates. It has even been said, that it takes about 10 years for a novice to become an expert programmer [32]. Educational research has been carried out to recognize the characteristics of novice programmers and to study the learning process and its connections to the different aspects of programming. Lately also differences between procedural and object-oriented education approaches have been studied, as Java and C++ have become common educational languages. We will now explore these issues more closely.

2.1 Characteristics of novice programmers

By definition, novice programmers lack the knowledge and skills of programming experts. Several different separating factors have been studied in the literature and were also reviewed by Robins et al. [27]. Common features for novices seem to be that they are limited to surface knowledge of programs and generally approach programming "line by line" rather than at the level of bigger program structures. Novices spend little time in planning and testing code, and when necessary, try to correct their programs with small local fixes instead of more thoroughly reformulating programs [17]. The knowledge of novices tends to be context specific rather than general [15], and they also often fail to apply correctly the knowledge they have obtained. In fact, an average student does not usually make much progress in an introductory programming course [17]. This was also noticed by the study of McCracken et al. [20], who noticed serious deficiencies in student's programming skills in introductory courses.

Milne and Rowe [21] studied in their recent survey the difficulties of C++ programming by conducting a web-based questionnaire for both students and teachers. One of the most obvious results was that students rated having less difficulties than deduced from teachers' answers. This was suggested to result from the fact that students believe themselves that they have understood the issue, but teachers see the remaining deficiencies in programming courseworks and examinations.

This supports the empirical observations of many teachers; programming novices often fail to recognize their own deficiencies.

Also the personal properties of the students affect their performance. There is no literature on significant differences in learning that would be caused by categories like gender or nationality, but general intelligence and mathematical or science abilities seem to be related to success at learning to program [19][4]. In a programming course, different student behaviours in confronting a problematic situation can be recognized. Perkins [25] named two main types: stoppers and movers. In problematic situation stoppers simply stop and abandon all hope of solving the problem on their own, while movers keep trying, modifying their code and use feedback about errors effectively. There are also extreme movers, "tinkerers", who cannot track their program, make changes more or less randomly, and like stoppers do not progress very much in their task.

All in all, there are effective and ineffective novices, i.e. students who learn without excessive effort and those who do not learn without inordinate personal attention [27]. Naturally, students' personal learning strategies and motivation affect their success in learning programming strategies. Robins et al. [27] stated that "Given that knowledge is (assumed to be) uniformly low, it is their preexisting strategies that initially distinguish effective and ineffective novices". Prior knowledge and practices can also be a major source of errors, especially when trying to transfer a step-by-step problem-solving solution directly from a natural language into a program [2]. The differences between the natural language and a programming language can easily cause problems. For example, some novices have understood that the condition in a "while" loop needs to apply continuously rather than it is tested once per iteration.

2.2 Different aspects of programming

Learning programming contains several activities, e.g., learning the language features, program design, and program comprehension. Typical approach in textbooks and programming courses is to start with declarative knowledge about a particular language. However, studies show that it is important to bring also other aspects to the first programming courses.

Several common deficits in novices' understanding of specific programming language constructs are presented in Soloway and Spohrer [29] and collected also by Pane and Myers [23]. For example, variable initialization seems to be more difficult to understand than updating or testing variables. Bugs with especially loops and conditionals are common, and actions that take place "behind the scene", like updating loop variables in "for" loops, are difficult for students. Students

have often many misconceptions in their understanding or implementing of recursion. Interestingly, novices were noticed to be more successful at writing recursive functions after learning about iterative functions, but not vice versa [14]. Expressions that are syntactically close to each other or mean different things in different contexts cause often practical difficulties, e.g. "123" and 123, or word "static" in C language. The survey by Milne and Rowe [21] showed that pointers seem to be very problematic, both according to the students and teachers. It was suggested that for this reason also dynamic memory allocation, reference parameters and other issues that need understanding of memory contents and pointers ranked high in the difficulty ratings. Also many other complex language features, like templates, casting, polymorphism and function overloading were considered difficult in the survey.

However, the main source of difficulty does not seem to be the syntax or understanding of concepts, but rather basic program planning [27]. It is important to distinguish between programming knowledge and programming strategies [6]. A student can learn to explain and understand a programming concept, e.g., what does a pointer mean, but still fails to use it appropriately in a program. Winslow [32] noticed that students may know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs. Even when they know how to solve the problem by hand, they have trouble translating it into an equivalent computer program. To help students to learn both concept knowledge and strategies for their use, Deek et al. [7] developed a problem-solving approach for a programming course. In their approach, language features were introduced to students only in the context to specific problems, little by little. This was shown to have positive effects on students' course results and their programming confidence. More discussion on problem-based learning in computer science courses can be found in Kay et al. [13].

Students have often great difficulties in understanding all the issues relating to the execution of a program. Du Boulay [10] stated that "... it takes quite a long time to learn the relation between a program on the page and the mechanism it describes." Students have difficulties in understanding that each instruction is executed in the state that has been created by the previous instructions. He stated that there should be a simple "notional machine", that simplifies the language and the machine so that all the program transformations can be visible. For example, LOGO language is an example of this kind of approach [24], and was commonly used in 80's in introductory programming education. It is based on the fact that all the instructions concern moving a turtle that is drawing on a display, thus making the program state and transformations clearly visible. However, nowadays most of the universities are required to teach programming languages that are

at least close to those used in the industry, and only rarely it is possible to select the approach for an introductory programming course based only on pedagogical reasons.

There is often little correspondence between the ability to write a program and the ability to read one. Programming courses should include them both. In addition, some basic test and debugging strategies should be taught [32]. Robins et al. [27] suggest that one more issue that complicates the learning of programming is the distinction between the model of the program as it was intended, and the model of the program, as it actually is. There are often mistakes in the design and bugs in the code. Also in working life, programmers face daily the need to understand a program that is running in an unexpected way. This requires an ability to track code to build a mental model of the program and predict its behaviour. This is one of skills that could be developed by emphasizing program comprehension and debugging strategies in the programming courses.

2.3 Object-oriented vs. procedural approaches

One of the claims for object-oriented approach has been that it is the natural way of conceptualizing real-world problems. However, studies do not seem to support that [27]. Rist even suggests that object-oriented programming adds the complexity of class structure to a procedural system [26]. Therefore, teachers using object-oriented approaches should not think that object-orientation is particularly easy for students, or that using OO from the very beginning relieves the teacher from teaching procedural programming issues.

Wiedenbeck et al. [31] studied students' comprehension of procedural and object-oriented small programs. They used programs that were functionally similar, but written either in Pascal/C or C++. Students in two comparison groups studied the programs and then answered questions that tested their understanding of different features and states of the program. The results showed that the distributed nature of control flow and "hidden" actions (e.g. constructor or destructor calls) made it more difficult for novices to form a mental representation of an OO program than of a corresponding procedural program. The class structure of the OO program made it a bit easier to understand program entities [30], but especially program flow and data flow issues were easier to understand from a procedural program.

The study by Milne and Rowe [21], recognized some concept difficulties relating to especially object-oriented paradigm. Both students and teachers considered constructors, virtual functions, and function overriding in inheritance among the most difficult issues in object-oriented programming. Interestingly, students saw virtual functions as only moderately difficult as opposed

to teachers who considered them as the second biggest problem among the concepts listed in the survey. Also, several respondents that were teaching an OO language stated that they believe that OO languages should only be approached once the student has a thorough grounding in procedural programming.

Barr et al. [1] investigated student programming errors in object-oriented Blue environment. Blue language and environment [16] is especially designed to ease students' learning of object-oriented paradigm with a simple language syntax and graphical support for classes and objects. However, also they noticed that students had more problems with higher-level object-oriented areas than syntax and style. Their findings support the already mentioned superficial approach typical for novice programmers; students could point out syntax and indentation errors in a given program but did not notice serious logical errors. This was also noticed in the inability of creating meaningful comments for the programs, often students just wrote a comment verbally describing the operation of a single code statement.

3. Visualization approaches for programming education

Visualizations have been used for long time in computer science education, since they have been considered as beneficial for understanding and learning the abstract and complex concepts of the field. Most of the approaches, however, concentrate on algorithm animation and less emphasis has been given to visualizing the basic structure of programs and their execution. However, since the research on algorithm animation has been carried out for a long time, there are results that can be utilized also in designing approaches for visualizing the basic programming structures.

3.1 Using and designing visualizations

A working group on Improving the Educational Impact of Algorithm Visualization studied the use of visualizations in computer science education with three surveys and an extensive literature review. The results of the work are published in Naps et al. [22]. Their work, although primarily aimed at algorithm visualizations, contain many interesting notions that can be applied also for designing and using visualizations in more general contexts. In fact, most of the recommendations of the working group seem to apply to the visualization approaches presented in the next subsection.

According to the surveys, all respondents were confident that visualizations help students' understanding and learning of concepts. Main statement against visualizations was the amount of time that is required to create visualizations, to install and study the technology involved, and to integrate the visualizations to the courses. The workgroup collected eleven best practices for good educational visualizations. Some of them are already included in the present approach of Codewitz materials, e.g. providing multiple views, flexible execution control and explanations on visualizations. Also the possibility for using input data given by the user was seen important, since it engages the user more to following the visualization. In addition to these, there were also some new proposals, e.g. "Support learner-built visualizations", "Support dynamic questions", and "Support dynamic feedback".

As seen from the mentioned best practices, the active engagement of students was considered very important. Naps et al. state "... visualization technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity." They proposed a taxonomy for different forms of learner engagement with visualization technology: 1. No viewing, 2. Viewing, 3. Responding, 4. Changing, 5. Constructing, 6. Presenting. The categories 2-6 can naturally also be combined. The first category means that there are no visualizations, and the last one means that students have a possibility construct a visualization that they should present and explain to the class. The steps in between describe different states of engagement in increasing order. Step 2 Viewing, including step-by-step execution and multiple views, is a basic requirement for all the higher levels. Examples of the steps 3-5 follow:

- 3. Responding. Present questions relating to the visualization, e.g. "What is the value of the variable a in line 3?"

- 4. Changing. Give student a chance to change the behaviour of the program, e.g. with different input. This can be also combined to the responding task, e.g. asking for an input to produce a certain output or to cover all possible execution paths.

- 5. Constructing. Offer students a possibility to create their own programs - possibly according to a given task - and see them visualized.

The strong belief of educators for the benefits of visualization was very interesting, since there are no experimental studies that support the educational effectiveness of visualizations. To support the further research on this field, the workgroup developed a research framework to evaluate the educational effectiveness of visualizations. The basic idea is first to design task specific questions, then use visualizations in teaching and then test student knowledge. When comparing to

another group with different teaching approach, it can be measured, whether visualizations have had positive effect to the learning. Similar approach could be used also in Codewitz-Minerva, when evaluating the quality and impact of the visualizations developed in the project.

3.2 Visualizations for basic programming concepts

Recursion was the only basic programming concept that has several visualization approached in the literature, e.g. [10] and [29]. In addition, there are many recursion visualizations that work on high level, not showing the actual code. Several tools have been developed to generally visualize program execution line-by-line, e.g. Teaching Machine [4], Thetis [11], and AnimPascal [31] with their traditional approach. PlanAni [30] is an example of a more recent work with an emphasis on the roles of the variables. However, in this section we concentrate on examples that have been developed to be used as teaching materials that introduce only few concepts at a time.

Rowe and Thorburn [28] developed a web-based tool called Vince to help students understand the execution of C programs. Their tool visualizes the workings of a C program step by step, showing the contents of the computer memory and providing user with explanations of each step. Tool display shows similar issues than present Codewitz materials, but more closely related to the actual execution model of the functions and the physical structure of the computer memory. The idea for the tool came from the program code debuggers, and this can be seen in the interface implementation. In addition to the example programs designed by the teacher, students can also provide their own programs or modify teacher's program, and then see how these programs are executed. The code examples are organized into prepared tutorials that students can study on their own. The study showed that Vince had had a positive effect on students' learning, and it was considered a good supplement for an introductory programming course.

Although not very graphical, another interesting work is published by Elenbogen et al. [11]. They have developed a set of interactive web exercises for learning various features of C++. Students can modify small program templates and check how their code affects the program behaviour. The questions are set so that they ask students about their understanding of the shown code segments, and students can execute the program themselves to find out the answers. On some exercises the execution is shown only by the results of the functions, but in the example of the classic Towers of Hanoi, also the animated towers are presented. Authors state to be satisfied with

their solution that enhances the learning in an introductory course. These applets can be also used as automatically assessed laboratory exercises even on large courses.

Nowadays, in the era of e-learning, great emphasis is given to the reusability and portability of learning materials. A concept of learning objects that cover a certain aspect of learning has been introduced and a final version of a standard description for learning object metadata was reached in 2002 [12]. This forms the basis for the working approach of Boyle and his team in London Metropolitan University [18]. Boyle [3] has been developing materials for visualizing certain programming concepts and describes them as pedagogically rich compound learning objects. Each object is designed towards a certain learning goal. The approach composes together several features visualizing the execution of e.g. a "while" loop and giving students questions and tasks relating to the concept. As opposed to larger visualization or learning support systems, the benefit of this approach is that this kind of objects can be easily taken in use and incorporated to education. Independent learning objects also make it possible to develop common repositories for reusable and portable learning materials. This is close to the basic principles of the whole Codewitz project.

4. Suggestions for the Codewitz project

Learning to program is a complex task and teachers should design their teaching approaches carefully. The traditional approach of concepts first is common and found to be effective, although also different approaches exist. There are plenty of typical novice programming errors, many of which are well collected into the technical report by Pane and Myers [23]. They present features and issues that should be taken into account when designing support systems for programming novices, and their work is useful also in connection to developing Codewitz materials.

However, the biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them. Robins et al. [27] suggest that teachers should focus more on combination and use of these features, especially on the underlying issues of basic program design. For example, building visualization examples of the basic structures and their combinations in different situations could promote students understanding of different strategies and build a mental "library" of different solution schemas for program design.

Comprehending program states has shown to be difficult already with procedural programming and seems to be even more so when using object-oriented paradigm. Therefore, visualizing the execution of small object-oriented programs and showing a full life cycle of objects,

would probably help many students. Since pointers and memory contents cause difficulties with many complicated object-oriented features, visualizations could be especially useful for illustrating program flow with these. Also the complicated inheritance features could be explained with dynamic visualizations, e.g., showing how virtual function calls are resolved or what happens when an object of a child class is instantiated.

Existing visualization approaches for basic programming structures are good examples also for Codewitz project. Also the research on algorithm simulation and visualization in educational settings can provide important information for developing Codewitz simulations. For example, the framework developed by Naps et al. [22] could be utilized when designing the research approach for evaluating educational effectiveness of Codewitz materials. Since large program execution visualization systems already exist, Codewitz should keep its concentration on developing independent small-scale examples. Small well documented educational entities (i.e. learning objects) that are easily usable and technically portable can be incorporated to the education in many ways without extensive training for teachers or students.

As opposed to the present materials, the partners of the Codewitz-Minerva project could also develop approaches that support more than just comprehension skills. Since learning problems are often connected to more advanced issues than individual concepts, visual applications could be directed to develop program generation, modification and debugging skills. If small examples, emphasizing few concepts at a time, could be developed to support students' active programming skills, they would also better engage the student in the learning situation. Since success in creating a functional program is a major positive force on students' traditional programming work, also visualizations could have more problem-solving nature instead of only representing concepts.

Finally, teachers should always remember that technical tools and visualizations are just learning aids and materials. It is necessary to consider educational issues when designing these technical aids, but however good the materials are, they can still be used either effectively or ineffectively. Teachers need to always thoroughly design their instructional approach to the issues on the course and how the aiding materials are incorporated to the education. Just giving a web address for students and advising them to use the Codewitz materials by themselves, will not reach all the students who would need the support for their learning. Material usage should be carefully designed and the settings should be well documented so that the project results will contain, in addition to the materials, also tested teaching strategies as examples of the use of them.

5. References

- [1] Barr, M., Holden, S., Phillips, D. & Greening, T. (1999). An exploration of novice programming errors in an object-oriented environment, *SIGCSE Bulletin*, 31(4), pp. 42-46.
- [2] Bonar, J. & Soloway, E. (1989). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers, in Soloway & Spohrer: *Studying the Novice Programmer*, pp. 325-354.
- [3] Boyle, T. (2003). Design principles for authoring dynamic, reusable learning objects. *Australian Journal of Educational Technology*, 19(1), pp. 46-58, available at <http://www.ascilite.org.au/ajet/ajet19/boyle.html>, referenced 2.12.2003.
- [4] Bruce-Lockhart, M. & Norvell, T. (2000). Lifting the hood of the computer: program animation with the Teaching Machine. *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, vol 2, pp. 831-835.
- [5] Byrne, P. & Lyons, G. (2001). The effect of student attributes on success in programming, *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pp. 49-52.
- [6] Codewitz project. <http://www.codewitz.net/>, referenced 2.12.2003.
- [7] Davies, S. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39, pp. 237-267.
- [8] Deek, F., Kimmel, H. & McHugh, J. (1998). Pedagogical changes in the delivery of the first-course in computer science: Problem solving, then programming. *Journal of Engineering Education*, 87, pp. 313-320.
- [9] Du Boulay. (1989) . Some difficulties of learning to program. In Soloway & Spohrer: *Studying the Novice Programmer*, pp. 283-300.
- [10] Elenbogen, B.S., Maxim, B.R. & McDonald, C. (2000). Yet, more Web exercises for learning C++, *Proc. of the 31st SIGCSE Technical Symposium on Computer Science Education SIGCSE Bulletin*, pp. 290-294.
- [11] Freund, S. & Roberts, E. (1996). Thetis: an ANSI C programming environment designed for introductory use. *Proc. of the 27th SIGCSE Technical Symposium on Computer Science Education*, pp. 300-304.
- [12] George, C. (2002). Using visualization to aid program construction tasks. *Proc. of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pp. 191-195.
- [13] IEEE Learning Technology Standards Committee. Final LOM Draft Standard, <http://ltsc.ieee.org/wg12/20020612-Final-LOM-Draft.html>, referenced at 2.12.2003.
- [14] Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J. & Crawford, K. (2000). Problems-based learning for foundation computer science courses. *Computer Science Education*, 10(2), pp. 109-128.
- [15] Kessler, C. & Anderson, J. (1989). Learning flow of control: recursive and iterative procedures. In Soloway & Spohrer: *Studying the Novice Programmer*, pp. 229-260.
- [16] Kurland, D., Pea, R., Clement, C. & Mawby, R. (1989). A Study of the development of programming ability and thinking skills in gih school students. In Soloway & Spohrer: *Studying the Novice Programmer*, pp. 209-228.

-
- [17] Kölling, M. & Rosenberg, J. (1996). Blue - A Language for Teaching Object-Oriented Programming, Proc. of the 27th SIGCSE Technical Symposium on Computer Science Education, pp. 190-194.
- [18] Linn, M. & Dalbey, J. (1989). Cognitive consequences of programming instruction. In Soloway & Spohrer: Studying the Novice Programmer, pp. 83-112.
- [19] London Metropolitan University. Learning Objects for Introductory Programming. <http://www.unl.ac.uk/ltri/learningobjects/index.htm>, referenced 2.12.2002.
- [20] Mayer, R., Dyck, J. & Vilberg, W. (1989). Learning to program and learning to think: what's the connection? In Soloway & Spohrer: Studying the Novice Programmer, pp. 113-124.
- [21] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, SIGCSE Bulletin, 33(4), pp. 125-180.
- [22] Milne, I., Rowe, G. (2002). Difficulties in Learning and teaching Programming - Views of Students and Tutors, Education and Information Technologies, 7(1), pp. 55-66.
- [23] Naps, T.L., Röbling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Velázquez-Iturbide, J.Á. (2003), Exploring the role of visualization and engagement in computer science education, SIGCSE Bulletin, 35(2), pp. 131-152.
- [24] Pane, J. & Myers, B. (1996). Usability Issues in the Design of Novice Programming Systems, School of Computer Science Technical Reports, Carnegie Mellon university, CMU-CS-96-132, available at <http://www.cs.cmu.edu/~pane/ftp/CMU-CS-96-132.pdf>
- [25] Papert, S. & Solomon, C. (1989). Twenty things to do with a computer. In Soloway & Spohrer: Studying the Novice Programmer, pp. 3-27.
- [26] Perkins, D., Hanconck, C., Hobbs, R., Martin, F. & Simmons, R. (1989). Conditions of learning in novice programmers. In Soloway & Spohrer: Studying the Novice Programmer, pp. 261-279.
- [27] Rist, R. (1996). Teaching Eiffel as a first language. Journal of Object-Oriented Programming, 9, pp. 30-41.
- [28] Robins, A., Rountree, J. & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion, Computer Science Education, 13(2), pp. 137-172.
- [29] Rowe, G., Thorburn, G. (2000). VINCE - an on-line tutorial tool for teaching introductory programming, British Journal of Educational Technology, 31(4), pp. 359-11p.
- [30] Sajaniemi, J. & Kuittinen, M. (2003). Program Animation Based on the Roles of Variables. Proceedings of the ACM 2003 Symposium on Software Visualization, pp. 7-16.
- [31] Satratzemi, M., Dagdilelis, V. & Evagelidis, G. (2001). SIGCSE Bulletin, 33(2), pp. 137-140.
- [32] Soloway, E. & Spohrer, J. (1989). Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale, New Jersey. 497 p.
- [33] Tung, S.-H., Chang, C.-T., Wong, W.-K. & Jehng, J.-C. (2001). Visual representations for recursion, International Journal of Human-Computer Studies, 54(3), pp. 285-300.
- [34] Wiedenbeck, S. & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles, International Journal of Human-Computer Studies, 51(1), pp. 71-87.

[35] Wiedenbeck, S., Ramalingam, V., Sarasamma, S. & Corritore C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11(3), pp. 255-282.

[36] Winslow, L.E. (1996). Programming pedagogy - A psychological overview. *SIGCSE Bulletin*, 28(3), pp. 17-22.