Our Experience Teaching After-School Programming to Parents and Their Children

Rudolf Pecinovský[1], Jiří Kofránek[2],
[1]University of Economics, Department of Information Technologies, Prague, Czech Republic
[2]Charles University in Prague, Laboratory of Biocybernetics, Prague, Czech Republic

Abstract - *Many years ago, Prague's sport clubs introduced courses of swimming, gymnastics, trekking as well as other physical training for parents and their children, in which parents and children are active participants who carry on the selected activities together. Compared to this thoroughly-explored problem, courses of programming for adults and children face a very different set of problems. We were inspired by the experience of after-school athletes and decided to examine how an interest group of after-school programming for parents and their children might operate. This paper summarizes a two-year experience with running such courses.*

Keywords: Education, Program Architecture, Object-Oriented Programming, Design Patterns, UML

1 Itroduction

The authors of this paper teach a wide range of students, from children attending interest groups, to regular lessons in middle schools and universities, and all the way up to running continuing education courses for professional programmers. They teach according to the time-tested methodology of *Architecture First* [14, 15] in all courses. But each kind of course has its unique needs and requires solving particular types of problems:

- Although children are immensely perceptive and acquire the presented items very quickly, they easily are drawn by their schoolmates from programming to playing computer games.
- University students usually divide quickly into two groups:
  - Those who have never programmed before, and need a slower start in which the basic programming constructions are presented first. However, as soon as they become acquainted with them, they may successfully proceed in the course, assuming their diligence and willingness to develop required programs.
  - Those who have already been taught programming consider the initial presentation of basic terms unnecessary, but don´t realize the subject matter differs significantly from what they learned in their school courses. Most often they bring a false idea from school that programming in an object language means object-oriented programming. They don´t realize that they learned to write only classically structured programs in prior courses, and that using classes does not mean object-oriented programming. These students usually have problems in the more advanced phases of the course, and only later realize that the subject matter presented at the beginning was not as self-evident as they thought. Thanks to their passivity they did not learn all that they should have learned, and do not know what they need at the moment.
- Professional programmers fight with the aforementioned problems of advanced students even more intensely. These programmers are sent to our courses by their employers, who discover that graduates know a number of frameworks and programming tools, but are not able to propose a good architecture for more extensive projects. Programmers with such training know to implement entered interfaces (designed by others), but they do not recognize where in their proposal they should define their own interface.

The reason these two groups of advanced programmers share common properties is well-characterized by a constructivist theory of teaching [7], which shows that new findings are grafted on to previously-adopted knowledge. If these new items are not in accord with the old ones, the students are not able to remember them properly. They subconsciously modify new information according to their existing experience, and therefore they place into their memory something slightly different than was presented to them. This was our chief reason for using the *Architecture First* methodology.

2 Lessons of programming for parents and their children

Analyzing the aforementioned problems, we had an idea to borrow the model of sport clubs. These are Czech after-school clubs that began offering courses of swimming, gymnastics, trekking and other training many years ago, and which are attended by both parents and children. Both are active participants, who carry out the given activity together. Therefore we opened the first

experimental group of programming for parents and their children in the Technical Center of Children and Youth House, Prague. Our aim was:

- To clarify if this methodology can teach both children and parents. They have different needs: the children are pure beginners, while their parents are in most cases able to program a little bit, despite not being professional programmers.
- To analyze the difference in acceptance, when the same subject matter is presented to these two significantly different groups of students.
- To verify our presumption that in certain cases the children could help their parents interpret the first pieces of information gained in the initial phases of the course.
- To utilize the interest of parents to avoid the possibility that children could slide from programming to computer games, and to ensure that they would prepare for the following lessons with the necessary care.

3 The Architecture First methodology

The ***Architecture First*** methodology, which we use in our courses, comes out of the two following assumptions:

- From the fact that nowadays the major part of a programmer´s work is taken over by various code generators, and that the only area which has and will resist automation for a long time is the program´s architecture proposal.
- From the well-known pedagogical model of the Early Bird Pattern, which says [2, 3]: *"Organize the course so that the most important topics are taught first. Teach the most important material, the 'big ideas', first (and often). When this seems impossible, teach the most important material as early as possible."*

In other words: if you consider the art of proper architecture creation the basic knowledge with which students should leave their schools or programming courses, you have to teach it from the very beginning [16, 17]. Applying this methodology to programming, we present the material in our courses in four phases [15]:

1. In the first phase, the learner runs in an interactive mode in which all code is created by a generator included in the development environment.
2. In the second phase, the user is turned over to a text mode in which the students repeats subject matter of the first phase while learning to actually write the programs that were created by the generator in the first phase,.
3. Then, the students become acquainted with more demanding constructions in the third stage, which are behind the limits of the generator´s abilities. In this stage we still concentrate to architecture view of program and do not explain algorithmic construction but sequence of statements.
4. In the fourth stage, the students become acquainted with basic algorithmic constructions and learn to use them in their programs. In the final phase the students learn further important data structures and programming expressions.

The complete process is demonstrated in the graphic programs the students create. The students have at their disposal a very simple graphic library, and they program behavior for the devices illustrated in that library (they create elevators, cars going through complicated tracks, etc.). We verified that the students thus obtain very visual feedback when they see the constructions functioning.

3.1 Introduction to OOP in an interactive mode

In the first phase, we take advantage of the *BlueJ* development environment [1, 9]. We start from the very beginning with a non trivial project (see Figure 1) and we explain the basic object constructions as follows: objects, classes, interfaces, and interface inheriting. We explain their usage in a program proposal, and the students interact with these and many others in the interactive mode in which the students simulate one of the program´s objects. The students can immediately try everything what they propose.

Due to the fact that the student don´t deal with coding in the first stage, they are not distracted by syntactic rules of the particular language and can instead concentrate on architectonic principles. This enables us to demonstrate how such basic constructions—which the implementation of an interface is—operate immediately from the very beginning. We present a number of design patterns in the first phase of explanation, (such as *Utility class, Singletons, Null Objects, Enumerated Type, Servant,*

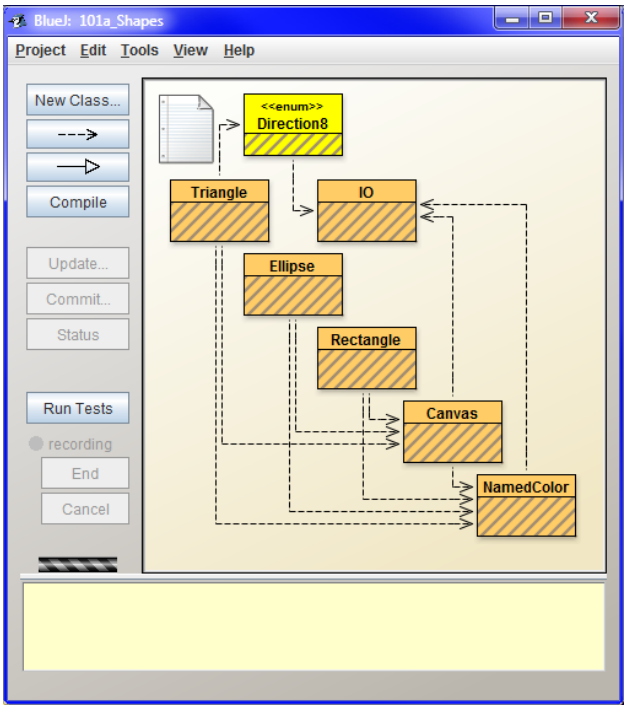

Figure 1: BlueJ with the starting project.

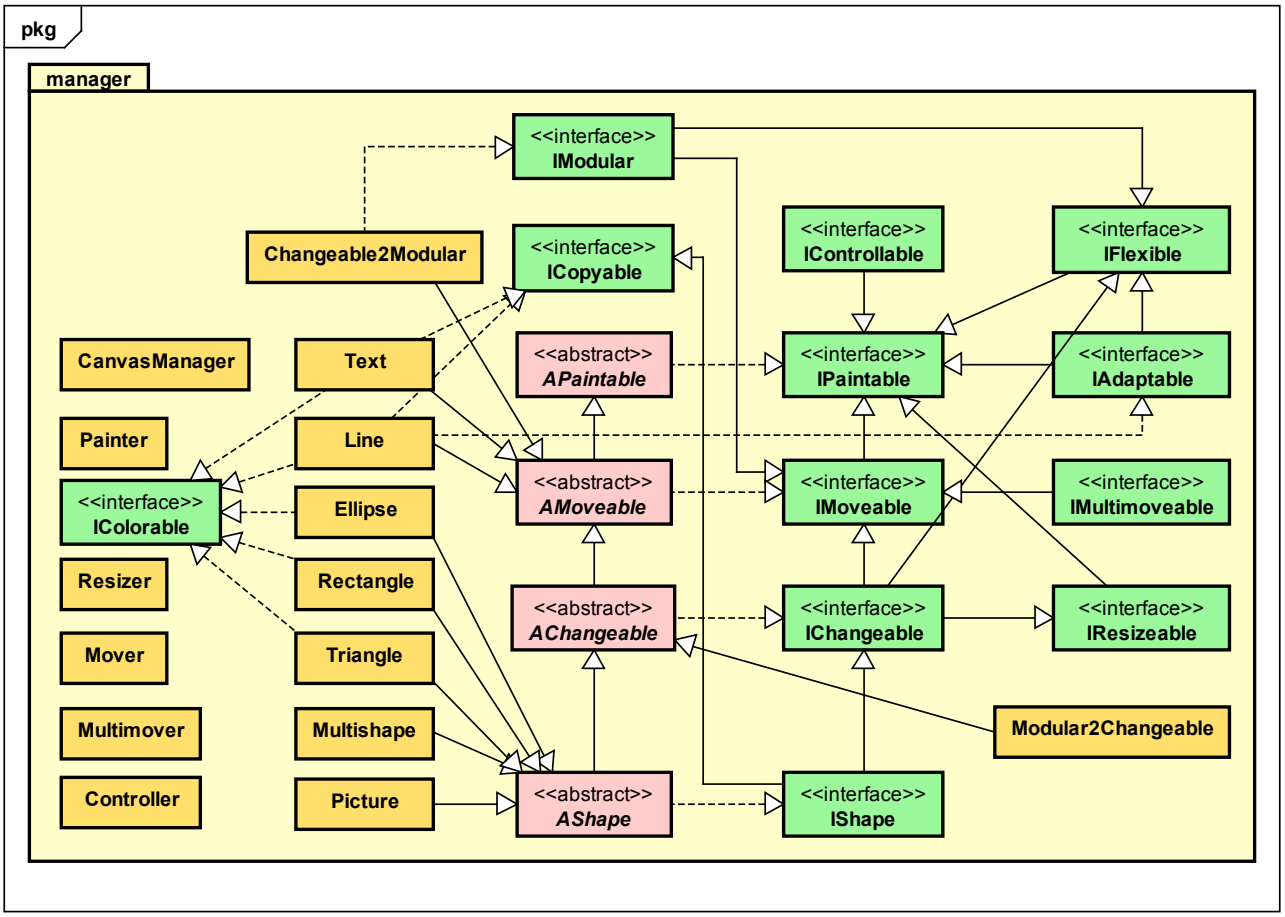

Figure 2: UML diagram of the project at the end of the first phase

Transfer Object - *Crate, Mediators, Observers* and so on) as well as certain architectural principles (programming against the interface, minimizing coupling, inversion of controls and similar concepts).

We frinish the explanation of the basic architectonic constructrions with a project roughly equivalent to the project at Figure 2.

3.2 Introduction to the syntax of the selected language

In the second phase of the couse, the students repeat the previous subject matter while learning to code the programs the development environment created in the first phase. Further, they learn how to use the syntactic rules and basic programming constructions of the selected language without being distracted with proposing a program. The program was already proposed in the first phase, and now they need only implement the rules of the first proposal.

The second phase finishes with an explanation of working with packages (name spaces), after which the students move to a professional development environment. For later phases, we used the *Java* language and the *NetBeans* development environment.

3.3 More complex programming constructions

In the third phase we leave the teaching environment and pass over to a professional environment which enables us to propose a more extensive project. Moreover, some of the students now become acquainted with some environments they will really use in their careers.

Further in this stage the students hear a deeper explanation of the architectural principles, interspersed with a presentation of a project using these programming constructions. And the introduction of design patterns continue: *State, Adapter* and *Decorator*.

In the third phase the students also encounter more complicated programming constructions, such as generic types, lambda-expressions, internal data types, collections and streams.

In this phase the students are programming more complex behavior: vehicles which are able to turn, or to drive along a zigzag track. They equip them with turn signals, and they also learn how to manage more cars at the same time. Yet I remind the reader that at this moment the students still don´t know algorithmic constructions like the conditional command and the cycle, because until now they have not needed them.

Until this moment, the students have made do with only interface, inheritance and their implementation. At the end of this phase, the students hear an explanation of inheritance, learning in which situations it can be used and, on the other hand, when they have to be careful with

it.

3.4 Algorithmic constructions

The fourth phase introduces algorithmic constructions which the students might encounter in older programs: conditional commands and cycles. At this time they can have a look at how the problems they solved without algorithmic constructions can be programmed differently.

Until now, we have used class diagrams as a primary means for program proposals. We taught the students to think primarily at the problem level and descend into the code level only at the moment when it is really necessary.

When explaining the algorithmic constructions we leave the *Unified Modeling Language (UML)* diagrams behind, and demonstrate all constructions through *kopenograms* [8, 24] (see Figure 3). *Kopenograms* are a handy tool for clear graphical representation of the structure of algorithms, and they have been particularly useful in teaching programming classes. They are a convenient supplement to the *UML* diagrams used to represent algorithmic structures.

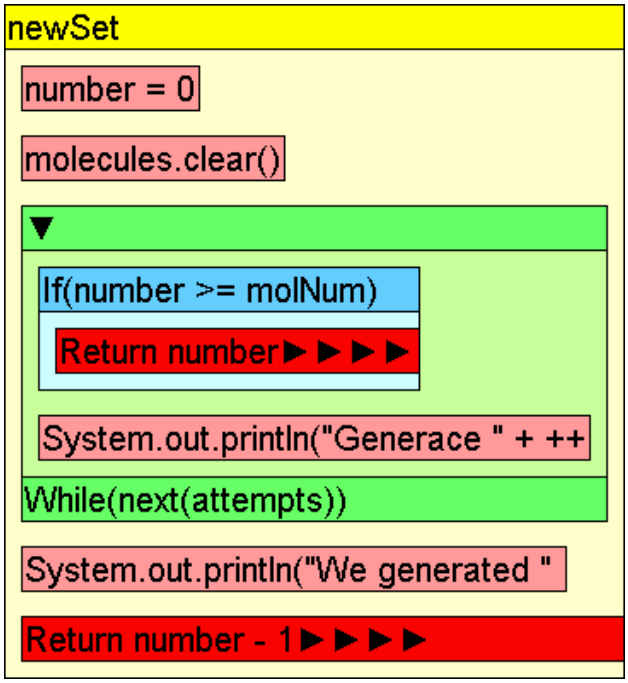

Figure 3: Kopenogram of a simple algorithm

3.5 Further data structures and program expressions

A number of useful data structures and programming expressions were not needed until this moment, but are necessary for future careers. These are presented in the final phase. These include: working with files, regular expressions, the basics of GUI applications, and some further useful items.

4 Results

So far, we have run two one-year terms. The courses were attended by children aged from 10 to 14 together with their parents. We didn´t offer a second year of teaching to the first class, because we wanted to reopen the first class and further examine our methodology. The couples continued studying according to the textbooks we provided, and from time to time they contacted us with a request for consultation.

During the one year course, we worked through the end of the third phase with the children and their parents. Eight pairs from the original twelve finished the course at the end of the first year, and ten couples out of twelve completed the course in the second year.

The course of teaching confirmed a lot of our assumptions, but also brought certain surprises.

4.1 Graphic interactive developing environment

The courses confirmed the utlity of the simple *BlueJ* interactive environment. This accorded with our previous experience, as well as experience of other authors [20], that it is useful to visualize relations among objects and classes through *UML* at the beginning, and only later pass over to a more complicated professional development environment. It is advantageous when the students get accustomed to using *UML* as a natural part of the problem analyses from the very beginning. We proved that using graphical tools for analysis and describing of the developed program is quite natural for children.

4.2 Speed of understanding

We verified that the same lessons can be used for both children and their parents. It is however important to avoid tedious lecturing, and instead stress the equivalencies between developed programs and a simulated reality. Students thus regard the objects in the program as natural equivalents of their patterns in the real world, not as an abstraction divorced from reality.

In several situations we experienced that the children really corrected their parents when they understood the subject matter incorrectly. The children explained the concepts in their own words, and the parents accepted their interpretation.

On the other hand, there were also children in the course (and surprisingly these were the older children) who always waited to see how their parents would understand the problem. They did not program anything independently and they only copied programs from their parents' computers

4.3 Unexpexted probles

There was also a surprise for us in these combined courses. When we taught courses attended only by children, we had no fundamental problems with homework. But these problems did occur in combined courses. The parents were not accustomed to doing homework, but instead learned how to justify the fact that they had not

completed their homework. And of course, the children quickly adapted to this and the general speed of progress in the course considerably decreased.

5. Discussion

Special languages are often used when teaching the basics of programming for children—as are special development environments connected with a graphic user interface, in which you move or draw with a little turtle (the language *Logo* [11], [12], [19]) or you manage a robot's movements (the language *Karel* [13], [4]). And so on. These visual teaching tools are brilliant if you want to teach children algorithmization and the basics of structured and modular programming. These visualization tools are sometimes also used in introductory programming courses at universities [5]. But they are insufficient for teaching more complicated programming constructs and principles of object architecture. Usually the development environment of the particular programming language is used for further teaching.

At this point, the students must make a great effort to learn the syntax of the given programming language and to start working with the development environment, and only on the basis of this knowledge can they learn more complex concepts of object architecture using the design patterns for creating their programs. However, the biggest problem of novice programmers seems to be, not understanding architectural concepts, but rather learning to apply them [10].

One way to overcome the requirement of syntax knowledge is using visualizing tools which enable users to compose and test object programs visually, without coding them. These tools must be sufficiently simple and cannot discourage the beginners by their complexity. One such tool is the aforementioned *BlueJ* environment [23]. The development of *BlueJ* was started in 1999 by Michael Kölling and John Rosenberg at the Monash University, as a successor to the *BlueJ* system. The *BlueJ* system was an integrated system with its own programming language and environment. *BlueJ* implements the Blue environment design for the Java programming language. BlueJ is currently being maintained by a joint team at the University of Kent (Canterbury, England) and La Trobe University in Melbourne, Australia.

In March 2009, the BlueJ project became free and open source software, licensed under the GNU GPL with the class path exception.

Nearly one thousand schools and universities are now using the *BlueJ* environment [23]. A textbook on this environment was published in its fifth edition last year [1] and has been translated into six national languages.

Inspired by the success of *BlueJ*, Microsoft implemented a similar environment into the Visual Studio 2005 development environment under the name of *Object Test Bench* [25] in 2005. At the same time, Microsoft submitted a patent registration [6] concerning their *"facility for testing an object in an integrated development environment without providing source code or knowing semantics of a language"*.

The patent registration caused a stormy reaction among *BlueJ* authors as well as users, and in the end no patent was awarded. *Object Test Bench* was also a part of *Visual Studio 2008*; however, this tool has not appeared in later versions of *Visual Studio*. Microsoft explained the removal by saying that majority of professional programmers use more complicated visual tools that are included in *Visual Studio*, and *Object Test Bench* was determined more useful for academic circles and teaching of programming.

Nevertheless, we believe that it´s a pity *Object Test Bench* cannot be downloaded into the new *Visual Studio*, even as an optional supplement for teaching *C#*. According to our experience, it is advantageous when we have a visualizing tool with a relatively simple interface at the user's disposal for teaching programming. This is because we don´t have to bother students at the beginning with the quite complicated task of managing a development environment. Students in our courses start with working in the simple *BlueJ* environment, and only later pass over to the professional *NetBeans* development environment. Learning and managing a more complex development environment (as well as necessary knowledge of programming language syntax) is far easier when the student has proper habits from a simple object development tool.

Consistently exploiting the possibilities of the simple *BlueJ* environment enabled us to teach basic principles of architecture to students even before we started to explain the more detailed presentation of syntactic structures of the language. Especially with children (who are not burdened with any previous knowledge of programming language) we proved that object-oriented thinking when creating architecture is very natural. Children understood the basic concepts, and clarified their understanding through practical examples using the *BlueJ* interactive mode.

When composing the tasks, we often program a particular part of the task in advance (these parts use constructions not explained yet),. This enables us to solve even more complicated tasks, and the computer solution motivates our students. Thus we could differ from classic textbook tasks, where the subject matter is demonstrated in simple tasks whose main purpose is to illustrate certain explained characteristics of the language. The typical task of a programmer is usually not to propose the solution of a simple problem, but on the contrary, to solve a current, usually very complicated program, proposed by somebody else, by certain new function(s). Therefore, our attitude was much closer to real programmer´s practice. Besides that, in many examples we could better demonstrate not only the properties of separate language construct, but combining and using various language constructs. The importance of this aspect is stressed by Robins et al. [18], who recommended in their study that instructions should focus not only on learning the new language features, but also on combining and using these features. His study also suggested that programming strategies should

receive more and more explicit attention in introductory programming courses.

Proceeding by short steps during the presentation proved to be very important, as did making an effort to see that the explanation and the programming tasks would followed naturally from the previous ones. As Winslow pointed out in *"Programming pedagogy"* [22], a good pedagogy requires the instructor to keep initial facts, models and rules simple, and only expand and refine them as the student gains more experience. Vihavainen [21] proposed a teaching methodology in which "small goals" are discussed as part of the teaching approach. "Small goals" are defined as the small parts that clearly set intermediate goals.

The result of gradually-expanding small goals was that the children succeeded in mastering quite a complicated course of object programming during one year, a course that can be compared in its content and range with some university courses.

6. Conclusions

The majority of programming courses begin with a great effort to explain the syntax of the selected programming language, as well to explain the meaning of separate language constructs. Only then do they proceed to an explanation of object architecture concepts and using design patterns. However, we are convinced that it is possible to turn this on its head, and start with an explanation of basic architectural concepts immediately, then teach the particularities of syntax throughout the whole course.

Our courses of programming for children with their parents confirmed our assumptions. They proved that when combined with suitable visualizing tools (such as BlueJ), it is possible and, and indeed better to teach object architecture from the very beginning of programming education.

7 References

[1] Barnes, D. J., Kölling, M. "*Objects first with Java: A practical introduction using Bluej*", 5th edition. Pearson Prentice Hall. ISBN 978-013-249266-9, 2012

[2] Bergin, J. "Fourteen Pedagogical Patterns"; *Proceedings of Fifth European Conference on Pattern Languages of Programs (EuroPLoP 2000)*, Irsee, Germany, 2000 [online] http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2000/2000_Bergin_FourteenPedagogicalPatterns.pdf.

[3] Bergin, J. "*Pedagogical Patterns: Advice For Educators*". CreateSpace Independent Publishing Platform, ISBN 1 4791 7182 4, 2012

[4] Bergin, J., Stehlik, M., Roberts, J., Pattis, R. "*Karel J Robot: A gentle introduction to the art of object-oriented programming in Java*". Dream Songs Press, 2005.

[5] Dai, K., Zhao, Y., Chen, R. "Research and Practice on Constructing the Course of Programming Language"; In *Computer and Information Technology (CIT), 10th IEEE International Conference on Computer and Information Technology*, pp. 2033-2038, 2010.

[6] Goenka, G., Das, P. P., Unnikrishnan, U. "*U.S. Patent Application 11/255,066*", 2005

[7] Harasim, L. "*Learning Theory and Online Technologies*". Routledge, Taylor & Francis Group, 7625 Empire Drive, Florence, KY 41042, ISBN 978 0 415 99976 2, 2011

[8] Kofránek, J., Pecinovský, R., Novák, P. "Kopenograms – Graphical Language for Structured Algorithms"; *Proceedings of the International Conference on Foundation of Computer Science. WorldComp 2012*, Las Vegas. CSREA Press. ISBN 1 601 32211-9, pp. 90-96, 2012

[9] Kölling, M., Quig, B., Patterson, A., Rosenberg, J. (2003)."The BlueJ system and its pedagogy"; *Computer Science Education*, Vol. 13, No 4, pp. 249-268, 2003

[10] Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. "A study of the difficulties of novice programmers"; *ACM SIGCSE Bulletin*, Vol. 37, No. 3, pp. 14-18, 2005

[11] Papert, S. "Teaching Children Thinking"; *Programmed Learning and Educational Technology*, Vol. 9, No. 5, pp. 245-255, 1972

[12] Papert, S. "*Final report of the Brookline LOGO Project*". Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1979.

[13] Pattis, R. E. "*Karel the robot: a gentle introduction to the art of programming*". John Wiley & Sons, Inc., 1981

[14] Pecinovský, R. "Using the methodology Design Patterns First by prototype testing with a user"; Proceedings of IMEM, Spišská Kapitula, pp. 1-10, [online] http://edu.pecinovsky.cz/papers/2009_IMEM_Using_DPF_by_testing_with_a_user.pdf, 2009.

[15] Pecinovský, R. "Principles of the Methodology Architecture First"; *Objekty 2012 – Proceedings of the 17th international conference on object oriented technologies*, Prague, pp 1-5, [online] http://edu.pecinovsky.cz/papers/2012_OB_ArchitectureFirst.pdf, 2012.

[16] Pecinovský R. "*Java 7 – Textbook of object oriented architecture for beginners*"; (in Czech), Grada, Prague, 2012.

[17] Pecinovský R. "*Java 8 – Textbook of object oriented architecture for intermediates*"; (in Czech), Grada, Prague, 2013.

[18] Robins, A., Rountree, J., & Rountree, N. "*Learning and teaching programming: A review and discussion*"; Computer Science Education, Vol. 13, No.2, pp. 137-172, 2003.

[19] Singh, J. K. "*Cognitive Effects of Programming in Logo. A Review of Literature and Synthesis of Strate-*

gies for Research."; Journal of Research on Computing in Education, Vol. 25, No. 1, pp. 88-104, 1992.

[20] Van Haaster, K., Hagan, D. "Teaching and learning with BlueJ: An evaluation of a pedagogical tool"; In *Information Science+ Information Technology Education Joint Conference, Rockhampton, QLD, Australia*, pp. 455-470, 2004

[21] Vihavainen, A., Paksula, M., Luukkainen, M. "Extreme apprenticeship method in teaching programming for beginners"; In *Proceedings of the 42nd ACM technical symposium on Computer science education,* ACM, pp. 93-98, 2011.

[22] Winslow, L. E. "Programming pedagogy—a psychological overview". *ACM SIGCSE Bulletin, 1996,* Vol. 28, No 3, pp. 17-22, 1996.

[23] http://www.bluej.org

[24] http://www.kopenogram.org

[25] Microsoft MSDN Library. Object Test Bench [online] *http://msdn.microsoft.com/en-us/library/c3775d98(v=vs.80).aspx*

Acknowledgement

This paper describes the outcome of research that has been accomplished as part of research program funded by the Ministry of Industry and Trade of the Czech Republic. This research was funded by *grant number FR—TI3/869.* These results were obtained within a larger research project concerned with the methodology of teaching computer programming. This research has been supported by *ICZ Group (commercial partner)* and *University of Economics, Prague (academic partner)*.

email to corresponding author: kofranek@gmail.com