Principles of the Methodology *Architecture First*

Rudolf Pecinovský

ICZ, Na Hřebenech II 1718/10, 140 00 Prague 4, Czech Republic,
University of Economics, Prague
rudolf@pecinovsky.cz

Abstract. The set of tasks, which does not need to be solved by a programmer, because certain program can solve it, extends all the time. The area which still resists to automation is a design of a good architecture. Most of the current methodologies teach primarily how to write a program in some programming language. The methodology *Architecture First* turns the current procedures up and starts with teaching the object oriented architecture. The paper introduces this methodology and explains its principles.

1 Introduction

The set of tasks, which does not need to be solved by a programmer, because certain program can solve it, extends all the time. At first we transferred from the machine code to high-level programming languages, then we started to use extensive libraries and nowadays we use various code generators more and more often. The area which still resists to automation is a design of a good architecture.

Almost all of the current methodologies start the teaching of programming with an explanation, how to write a program in some programming language ([1]). Although the methodology *Object First* came with the idea to start teaching with an explanation of basic behavior of objects in the class diagram, its authors did not realize how brilliant this idea is and what implication it brings including the possibility to continue in explaining the architecture. Therefore, the authors continue in their textbooks with a classical approach and teach the syntax and coding. Despite the fact their development environment *BlueJ* offers a simple code generator and thus it allows to continue the explanation in the architecture level and to leave the definition of simpler programs upon the shoulders of used IDE, they almost do not take the advantage of this possibility in their textbooks.

Many authors showed in their papers (e.g. [4], [8]), that the programming style, which students learn as the first one, influences their work and style of designing for a long time. This is not valid only as far as the change of programming paradigm is concerned – e.g. of transfer from the structured programming to the object oriented one. We can observe the same effect when we begin teaching the students, who already know to develop the intermediate complex programs, how to design the architecture of these programs. When such students receive an assignment, they often slip down to thinking how to code a particular functionality. They subordinate the archi-

tecture of the whole project to the way of coding it into the used programming language.

2 The Basic Principles

The methodology *Architecture First* turns the current procedures up. It follows the pedagogical early bird pattern that says: *"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."* If we decide to teach primarily the architecture design, because we know, that the computer takes upon its shoulders the significant part of the subsequent coding, we should teach the basic architectonic principles as early as possible and teach the principles of coding only subsequently as one of the implementation ways of the designed architecture.

The methodology *Architecture First* explains firstly the basic principles of building architecture of the object oriented programs and only after the students absorb these principles it continues with the explanation of the way, how to code (implement) the designed program.

Some of you can oppose that a number of courses proceeds like that. But the main difference is that majority of courses remains in their explanation in the theoretical level, whilst the *Architecture First* methodology uses in the introductory explanation the interactive capabilities of the current IDE (*BlueJ*), especially its ability to create the program performing the shown activity. Thus the students can immediately carry out their designs.

2.1 Interactive Phase

In the interactive mode student acts as one of objects in the project. The student shows to IDE, how the designed program should behave. It sends messages to other objects (including the IDE) and the IDE than creates a program (defines a method), which can repeat the shown (demonstrated) action. The students work in a mode similar to the one used in some office programs for creating simple macros.

Due to the fact that in the first phase the students don't deal with coding the designed program, they are not diverted by syntax rules of the used language and they may concentrate above all on the explained architecture and its principles. This enables us to introduce such basic constructs as interface and its implementation already at the very beginning of the explanation – we teach it in the second lesson. In the first phase we can also explain many design patterns (*Utility class*, *Singleton, Null Object*, *Enumerated Type*, *Servant*, *Transfer Object (Crate)*, *Mediator*, *Observer* …) and demonstrate their usage as well as some architectonic principles (program to interface, inversion of control, DRY, minimizing of coupling, casting from child to parent etc.).

2.1 Revision Phase

In the second phase of teaching the students again go through the previous topics and learn how to implement programs they designed in the first phase and let the IDE to implement them. In this phase they learn the syntax rules and the basic programming constructs of the current language without being diverted by the design of the required program. The program has been already designed in the first phase and they now only revise the rules of its design.

This repeating is very necessary, because the students often do not remember properly the exact meaning of explained notions in the first phase. This problem occurs especially at students with previous programming experience, who often remember the provided information distorted, because the current information does not correspond exactly to their previous experience received during their work in a different paradigm and their brains subconsciously modify it to their experience.

2.1 Enhancing Phase

The explanation of architectonic principles is then further extended in the third phase. It is already interspersed with an explanation of the particular implementation of the needed programming constructs.

2 Design of Algorithms

In the OOP era of the present times, sometimes it tends to be forgotten that the design of rather complex algorithms may be ahead of us at the end of analysis. Several graphical languages are available for their representation. The most often used are flowcharts, UML activity diagrams and Nassi-Schneiderman diagrams. However, all of them have some drawbacks. Flowcharts and UML activity diagrams do not force using of structured algorithmic constructs. Nassi-Schneiderman diagrams use oblique lines, which degenerate the space for conditions of condition statements.

The experience proved the graphical tool *kopenograms* () as the most effective – a handy tool for clear graphical representation of the algorithms´ structure. In our experience kopenograms have found a long-term application particularly in teaching the programming classes. They are a convenient supplement of UML diagrams used to represent algorithmic structures.

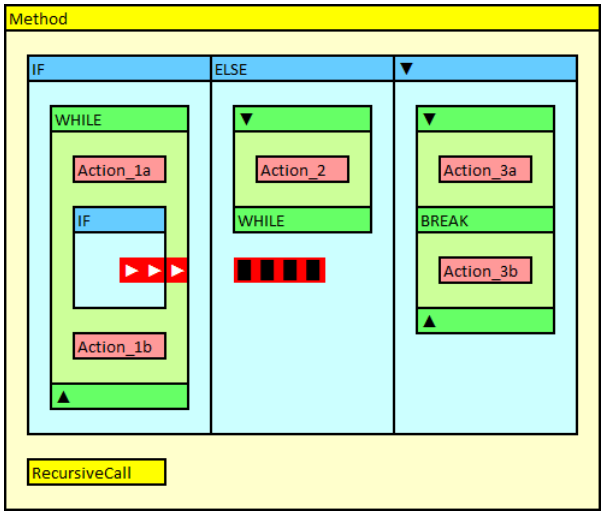

Fig. 1. Kopenograms

Acknowledgment

This paper describes the outcome of research that has been accomplished as part of research program funded the ICZ a. s. company.

References

[1] Computing curricula 2001. *Journal on Educational Resources in Computing*. Volume 1 Issue 3es

[2] Kofránek Jiří, Pecinovský Rudolf, Novák Petr: Kopenograms – Graphical Language for Structured Algorithms. *Proceedings of the 2012 International Conference on Foundation of Computer Science. WorldComp 2012* Las Vegas. CSREA Press. ISBN 1-60132-211-9

[3] Kopenograms home page: http://www.kopenogram.org

[4] Liberman Neomi, Beeri Catriel, Kolikant Yifat Ben-David: Difficulties in Learning Inheritance and Polymorphism. ACM Transactions on Computing Education. Volume 11 Issue 1, February 2011.

[5] Pecinovský R.: Early Introduction of Inheritance Considered Harmful. *Proceedings of Objects 2009*, Hradec Králové. ISBN 978-80-7435-009-2. DOI = http://edu.pecinovsky.cz/papers/

[6] Pecinovský Rudolf: Using the methodology Design Patterns First by prototype testing with a user. *Proceedings of IMEM 2009*, Spišská Kapitula. DOI = http://edu.pecinovsky.cz/papers/

[7] Pecinovský Rudolf: *Learn Object Oriented Thinking and Programming*. – Translation of the original book: *OOP – Naučte se myslet a programovat objektově*. Computer Press 2010. ISBN 978-80-251-2126-9. English version to be published.
[8] Robins A., Rountreek J., Rountree N.: Learning and teaching programming: A review and discussion. Comput. Sci. Educ. 13, 2, 137–172.