Hristov, H., „Review and outlooks of the means for visualization of syntax semantics and source code. Procedural and object oriented paradigm – differences", Anniversary International Conference REMIA 2010, Plovdiv, 2010, pp. 443-450, ISBN 978-954-423-648-9

REVIEW AND OUTLOOKS OF THE MEANS FOR VISUALIZATION OF SYNTAX SEMANTICS AND SOURCE CODE. PROCEDURAL AND OBJECT ORIENTED PARADIGM – DIFFERENCES

Hristo Hristov

Abstract. *In the article, we have reviewed the means for visualization of syntax, semantics and source code for programming languages which support procedural and/or object-oriented paradigm. It is examined how the structure of the source code of the structural and object-oriented programming styles has influenced different approaches for their teaching.*

We maintain a thesis valid for the object-oriented programming paradigm, which claims that the activities for design and programming of classes are done by the same specialist, and the training of this specialist should include design as well as programming skills and knowledge for modeling of abstract data structures. We put the question how a high level of abstraction in the object-oriented paradigm should be presented in simple model in the design stage, so the complexity in the programming stage stay low and be easily learnable. We give answer to this question, by building models using the UML notation, as we take a concrete example from the teaching practice including programming techniques for inheritance and polymorphism.

Keywords: Programming languages, Teaching methodology, Programming paradigm, Means for visualization.

Mathematics Subject Classification 2010: 68U02

1. INTRODUCTION

In general a program can be organized in two ways: around the code (what is happening) or around the data (with what it works). When structural programming is used, the programs are organized around the code i.e. "the code works on the data". Object-oriented programs work the other way. They are organized around the data, and the key feature is "data control access to the code" [1]. Procedural languages can be viewed as syntactic generalization of Neumann computers. Their semantics is the same as the semantics of the machine languages. Therefore procedural languages are high-level abstract versions of the von Neumann computers [2]. What follows from this fact is that focus of attention and research in programming languages is the algorithm, as the program in essence is carrying out of an algorithm. The increasing level of complexity of the algorithmic implementations has led to a search for means of their representation. Thus were created methodologies for teaching of procedural programming languages, which main focus is visualization of the algorithm, along with syntax and semantics of the language. The most used means for visualization, used in the specialized literature can be formal such as: *the Backus-Naur meta language* and *language of syntactic diagrams,* but can also be informal. In second case there is "non precision" from the point of exact definition and formalization, but this is justified as it is more accessible and easy to understand for students and learners. Unlike syntax, it is common the semantics of a programming language to be presented descriptively, and it is natural. In some cases it is appropriate the description to be enriched with logical block schemes, diagrams and examples. For instance the block schemes are useful when presenting conditional operators, cycle operators, calling subroutine, etc. Using logical block schemes and other kinds of diagrams makes it easy to understand the logics of the operator as well as it easy remembering.

Essential advancement in the development of the procedural programming is made with the development of the so called "Structured program theorem", connected with the names of Corrado Böhm and Giuseppe Jacopini. It states that every computable function can be implemented in a programming language that combines subprograms in only three specific ways: *„sequence"*, *„selection"* and *„repetition"*. With the gained view over the question, the methodological tasks when teaching a structured programming language have become much easier. For the presentation of *source code* it will be enough to find out means which represent the three outlined program constructions – separate or in combination with one another. Besides the source code of a given programming language, for ease of learning, the following means for visualization have gained popularity in practice: *logical block-schemes, text in form of pseudocode, Nassi-Shneiderman diagrams, HIPO diagrams, diagrams of data flows etc.* Such means are natural instruments for creation of methods for representing source code. But they have reached the boundaries of their usefulness and although they have their place during the lessons it is necessary to search for methods which use means corresponding to the contemporary technologies. Such means which *animate algorithms, dynamically visualize graphics, generate texts* and other solutions which can be integrated in the computer platforms. Of course, such methods are created daily from software companies and university teams. Such system for helping the whole process of teaching programming skills is being created at FMI in PU [3].

Deduction №1. Commonly used means for description and presentation of syntax, semantics and visualization of source code are: formal – Backus-Naur meta language, logical block-schemes, language of syntactic diagrams, Nassi-Shneiderman diagrams and informal – descriptive explanations, text in form of pseudocode etc. These means have reached the boundaries of their usefulness.

Basis for the development of new means for visualization of syntax, semantics and source code will be the technological solutions which dynamically visualize and automatically generate text, sounds, animation and video in such a way that an "intelligent" dialogue between the computer system and the student can be achieved.

2. BEHAVIOR OF THE PROGRAMMER WHEN WRITING STRUCTURAL AND OBJECT-ORIENTED CODE.

The object-oriented paradigm is created by combining the best ideas of the structural programming style with some new principles and conceptions. The result is different way of organization of a program, with higher level of abstraction.

How the high level of abstraction of the object-oriented programming languages influences the methodologies of teaching?

If we assume (with some notes), that the development of every software project passes the following stages: analysis, design, programming, test and support and ask what is the purpose of the programming language in such project we must note that the structural programming languages are *means for computer implementation of an algorithm,* while object-oriented languages are *means for computer modeling and implementation of real objects.* Indeed, in structural programming during the design stage a model of the problem is created and algorithms are chosen (or created), which during the programming stage are implemented. Said with other words programmers deal only with the programming; analysis and design is done by more experienced specialists – software architects.

Is this the same with object-oriented programming?

In object-oriented programming style the programmer takes part in both the design and programming stages. Classes and objects are basic concepts in the object-oriented paradigm and by definition for every program: *a class is an abstract data structure* which has its own characteristics and interface, while an *object* is concrete *implementation* of a class with its own memory, called state of the characteristics of the object and its own behavior, defined by the interface of the class, but depending on the current state of the characteristics. Therefore, *the class* is the blueprint from which from *the object* is created an entity; its creation before programming is designed - the activities *designing a class* and *programming a class* are closely connected and are done by the same specialist.

Deduction №2. In object-oriented paradigm the *design* and *programming* of classes is done by the same specialist, therefore the training of this specialist must include both design and programming knowledge and skills for modeling of abstract data structures.

3. DIFFERENCES IN THE WAYS OF TEACHING STRUCTURED AND OBJECT-ORIENTED PARADIGM.

In every textbook for procedural programming, we find unchangeably the following line of themes: *common structure of a program*, *basic variables and types, expressions, assignment operators, input/output operators, conditional and cyclic operators, composite types, functions, recursion, etc.* [4] The gained, during the years, clarity of what learning material follows which has made easy the tasks faced by the methodologies for teaching procedural programming style. On the other side, in object-oriented programming style there is no concrete and commonly accepted methodology for presenting the learning material. The question of what pedagogical approach to be used when introducing the ”object” concept is still open, with two positions being defended – early introduction and late introduction.

Why is accepted the early introduction of objects?

Object-oriented languages have the purpose to conduct concepts, and to learn and understand those concepts, one should know the principles which build them. The late introduction of objects means that structural way of thinking for programs should be used until objects are presented. This amplifies the algorithmic way of thinking and creates barriers, which make harder the understanding of the model of organization and functioning of an object-oriented program. Although the implementation of each method requires an algorithm, this is not main principle when writing a program. Beginning point for building software using object-oriented language are the principles – *abstraction, encapsulation, inheritance and polymorphism*. And if, from pedagogical point of view, *inheritance* and *polymorphism* are principles which are not necessary in the beginning lessons, *abstraction* and *encapsulation* are mandatory as a way of thinking and model for designing an abstract data structure. Separating those four principles from syntax and semantics for given object-oriented programming language has key role for the good teaching practice of this paradigm. It is necessary for the student to gain conceptual thinking how to *design* his programs based on the above mentioned principles, and then to program using the syntax and semantics of a given object-oriented language. All this requires early introduction to objects and classes in the learning process. Even more – guiding motif when composing the learning content should be the principles of object-oriented programming, not the grammar of the programming language. As in the ideology of object-oriented programming the first main principle is “everything is an object”[5], then with this should the teaching begin.

Deduction №3. Regardless of the methodology for teaching the procedural programming paradigm, it is common practice that *the structure of the learning content should follow the structure of the programming language.*

Deduction №4. The early introduction of the “object” concept indicates teaching of object-oriented programming style, the late introduction indicates mixed approach with structural way of teaching in the beginning and then adding the object-oriented principles.

Here arises another important methodological question! What means and methods should be used during the design stage such that they will allow to differentiate *design* from *programming* from one side, and to underline the dependency of the programming stage from the design stage? For the programmer and the student it is of big significance, how the high level of abstraction in the object-oriented paradigm should be presented through simple model in the design stage so that the complexity in the programming stage will not rise?

Reasonable answer to the first question is given by the Unified Modeling Language - UML, while answer to the second is the appropriate use of its graphical notation.

What follows is the design a part of simple program meant for teaching purposes, with which we will underline the power of the UML notation. The used example is not method for teaching, but it can serve as basis for the creation of methods for teaching. It is given as technique which uses graphical model as object for discussion between teacher and student.

So let put a task to create a program which shows how the principles of programming work – *abstraction, inheritance, polymorphism* and some of their techniques *composition, implementation, overriding*, *up casting* and others. For the purpose we will use a class which we will call *ComputerDevice.* We assume that devices from this type are: *Laptop*, *MobilePhone* и *GPS-Device,* for which we will also create classes. Before designing the above mentioned, in order to avoid some difficulties which may arise in the future and to follow the good practice in object-oriented design, we will define common for all other classes – the class *Device*. After that we design the concrete class – *ComputerDevice*. Finally we look at the devices: *Laptop*, *MobilePhone* and *GPS-Device.* The first principle of object-oriented programming that we come upon is *abstraction* which by definition includes taking those characteristics of the real world object which are relevant to the task and ignoring the ones that have no relation to the problem. Before doing this let's define two interfaces: first - *Action*, which has methods *open()* and *close()* and second - *Performance*, with methods *play()* and *stop().* As the operations *open()* and *close()* in the context of our task are more common than *play()* and *stop()*, we design the interface *Performance,* to be inheritor of the *Action* interface. To show this connection we use UML generalization as shown on *(Fig. 1).*

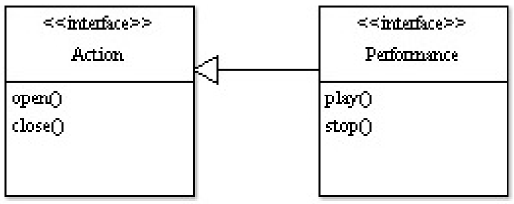

Fig.1. Interfaces presented by UML class diagram and generalization notation

Now back to classes which we bind in hierarchy. As expected base for the hierarchy is the *Device* class, its inheritor is *ComputerDevice;* at the bottom of the hierarchy are *Laptop*, *MobilePhone* and *GPS-Device.* Besides this we make *ComputerDevice to* implement the *Action* interface*,* and *Laptop* and *MobilePhone* classes to implement the *Performance* interface. But *Performance* is inheritor to *Action,* therefore in *Laptop and MobilePhone* may be implemented the methods of the *Action* interface – *open() and close().* From other side these methods are implemented in the parent class – *ComputerDevice*. Not such is the case with *GPS-Device* class*,* as it does not implement the *Performance* interface. *GPS-Device* inherits *ComputerDevice* and therefore can predefine the methods of its parent, but if it does not need them this is not obligatory. Such relations of inheritance and implementation between interfaces and classes presented in the programming class may sound difficult to understand for the student at first, but everything comes into place, when we view closely and discuss the following *(Fig. 2)* (Let's underline that the figure is used for discussion, and not for a model to be used for writing source code).

Although, that from the UML model we cannot tell what the source code should be, we can still have a rough idea how it should look like and we will have concept for its structure.

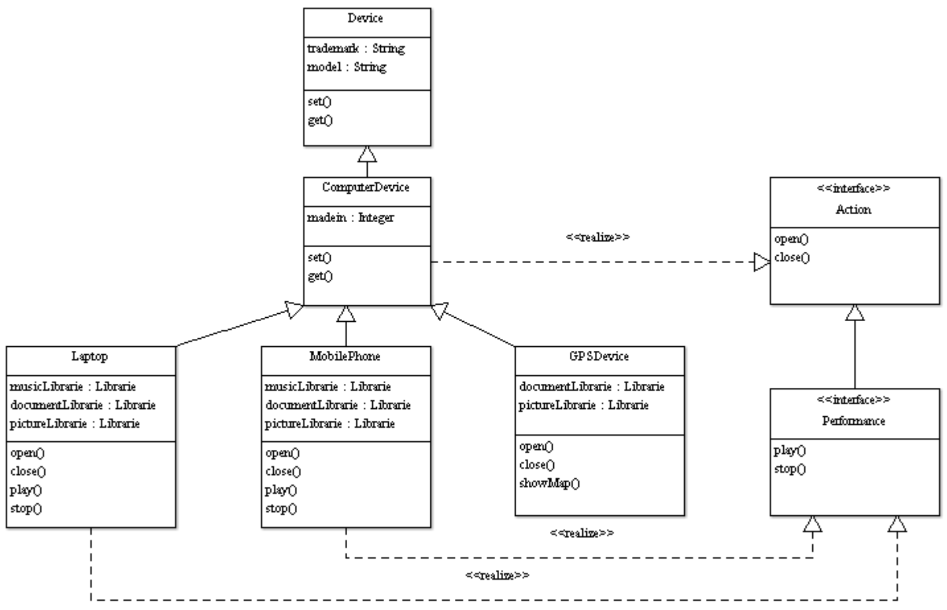

Fig. 2. Model of hierarchy of classes and interfaces

Similar model as *(Fig. 2)* is good to show the techniques *expanding* of classes and interfaces during inheritance and *overriding*, *run-time banding* and *up casting* in polymorphism. But what is the situation when these techniques should be interpretated into source code? Independent of how clearly the thesis is stated, the abstraction of the model turns out to be high barrier for most of the students. Here comes the advantage of UML notations – using small number of graphical signs and diagrams we can summarize hundreds of lines source code. In most cases the whole graphical model is visible on the screen and thus it easy to trace relations like composition, inheritance, implementation, overriding, etc. Thus the tidy and easy to understand graphical model becomes good basis for communication between the teacher and the student. Such communication has very important pedagogical characteristic – the discussion is at level *design* and *modeling* and *not programming.* It was mentioned that in the first case the discussion is about abstract principles and techniques of programming, in the second it is about syntax and semantic rules of concrete programming language. Such differentiation separates the levels of abstraction and has significant meaning for the student.

Besides, by creating models of UML notations one can have a rough idea of how the source code should look like which helps the student to get clear view of the software perspective.

Deduction №5. By the use of graphical notation like the UML language, one can achieve differentiation of the *design* and *programming* stages and separation of the levels of abstraction. Carefully designed graphical models create good basis for wholesome discussions between teacher and student.

4. CONCLUSION.

The above review and analysis of the means for presentation and their separation of those who visualize procedural source code and the ones that visualize object-oriented one shows that in the first case it is enough to visualize the algorithm of the program, and in the second – it is necessary to visualize the model of the program's structure. The commonly accepted means for visualization of algorithms, such as logical block schemes, syntactic diagrams, Nassi-Shneiderman diagrams, pseudo code, etc., have reached the boundaries of their usefulness. Their use is necessary in the programming classes, but the perspectives for improvement of the methodologies for teaching are connected with technological solutions which dynamically visualize and automatically generate text, sound, animation, graphics and. In the second case, object-oriented programming, the problem of visualization is more abstract and it is necessary to be split in two: first, design – the model of the program is visualized, as main feature of the paradigm is the relations between objects and second visualization of a source code in the boundaries of a class method – viewed as realization of an algorithm. The separating of visualization at level *design* and level *programming* helps to reduce the complexity of the problem. It is perspective to use graphical notations, through which simple but conceptual models of the programs are created. Such models give opportunity to discuss the advantages and disadvantages of using one or another programming technique in given situation.

REFERENCES

[1] Schildt, H. A Beginner's Guide, Sofia, SoftPress Ltd, 2001.
[2] Todorova, M. Programming in C++, Sofia, Ciela, 2002.
[3] Krushkov, H. M. Krushkova. A Computer-based Tutoring System for Programming, Mathematics and Education in Mathematics. Proceedings of the Thirty Ninth Spring Conference of the Union of Bulgarian Mathematicians, (2010), pp. 354-358.
[4] Dobrev, D. Methodology of Statement Control and Data Choice in Procedure-Oriented Programming, Mathematics and Education in Mathematics. Proceedings of the Thirty Fifth Spring Conference of the Union of Bulgarian Mathematicians, (2006), pp. 387-392.
[5] Eckel, B. Thinking in Java, Sofia, SoftPress Ltd, 2002.

Hristo Hristov
Faculty of Mathematics and Informatics
Plovdiv University "Paisii Helendarski"
236 Bulgaria Blvd.
4003 Plovdiv, Bulgaria
e-mail: hth@uni-plovdiv.bg

ОБЗОР И ПЕРСПЕКТИВИ НА СРЕДСТВА ЗА ОНАГЛЕДЯВАНЕ НА СИНТАКСИС, СЕМАНТИКА И ПРОГРАМЕН КОД. ПРОЦЕДУРНА И ОБЕКТНО ОРИЕНТИРАНА ПАРАДИГМА - РАЗЛИЧИЯ

Христо Тошков Христов

Резюме. *В статията е направен обзор на средства за онагледяване на синтаксис, семантика и изходен код на езици за програмиране, които поддържат процедурна и/или обектно-ориентирана парадигма на програмиране. Разгледано е, как организацията на програмния код при структурния и обектно-ориентирания стил на програмиране е повлиял по различен начин върху подходите за тяхното преподаване.*

Застъпена е и защитена теза валидна за обектно-ориентираната парадигма на програмиране, в която се твърди, че дейностите по проектиране и програмиране на клас се извършват от един и същ специалист и следователно обучението на този специалист трябва да засяга както проектантски, така и програмистки познания и умения за моделиране на абстрактни структури от данни. Поставен е въпроса за това, как високото ниво на абстракция в обектно-ориентираната парадигма да се представи, чрез прост модел в етапа на проектиране, така, че сложността в етапа на програмиране да не се увеличава, а от тук и трудностите при нейното изучаване да намаляват. Отговор на въпроса се дава, чрез създаването на графични модели посредством UML нотация, като е разгледан конкретен пример от преподавателската практика засягащ техники на програмиране обхващащи концепциите наследяване и полиморфизъм.

ФМИ при ПУ „Паисий Хилендарски"
гр. Пловдив, п.к. 4003, бул. България 236
e-mail: hth@uni-plovdiv.bg, hristo.toshkov@gmail.com