On the Complexity of Class

Ákos Fóthi, Judit Nyéky-Gaizler, and Zoltán Porkoláb

Department of General Computer Science
University Eötvös Loránd, Budapest, Hungary
fa@ludens.elte.hu, nyeky@ludens.elte.hu, gsd@ludens.elte.hu

Abstract. Object-oriented programs are constructed with the help of the same control structures as traditional ones. At first sight, therefore, their complexity can be measured the same way as the complexity of the traditional programs. In this case the complexity depends on the nesting level of the control structures, as it has been shown by Piwowarski, Harrison, Magel, Howatt, Baker etc. [5],[3],[4],[8]. Why do we still have the feeling that object-oriented programs are more simple than the traditional ones? To answer this, we have to introduce a new measure of complexity. The measures mentioned above have a common problem: each of them evaluates the complexity of a program only from the point of view of its control structure. Our opinion discussed here is that the complexity of a program is a sum of three components:
(1) the complexity of its control structure,
(2) the complexity of data types used,
(3) the complexity of the data handling (ie. the complexity of the connection between the control structure and the data types).
We give a suggestion for the measure of complexity of a program. First we show this measure working well on procedural programs. Then we extend the measure to object-oriented programs too. This new measure of complexity is used to argue why good object-oriented programs could seem more simple.

1 Introduction

There are several methods of measuring program-complexity. The complexity of programs depends on the number of operators and operands (the software science measure); on the number of predicates (cyclomatic complexity); but these measures do not characterize sufficiently the nature of complexity, since n nested loops or n nested if statements are undoubtedly more complex than the sequence of n loops, or the sequence of n decisions. The complexity of programs was so far mostly measured on the basis of its control structure.

Nowadays, one of the most frequently read notion in the literature of programming methodology is the 'object-oriented' one. While constructing great systems the questions of reusability and extendibility became of key importance. The more simple a program is the easier it is to understand, later to modify or reuse some parts of it in the case of the construction of other similar programs. Followers of object-oriented methodology state that professional software production becomes notably simplified using this new technique, which results in enormous cost decrease.

Object-oriented programs contain the same control structures (sequence, if- and loop statements) as the traditional ones, thus there seems to be no difference in their complexity.

What is the greatest novelty of this design approach? It draws the attention to the importance of precise definition and consistent use of abstract data types. Actually, if we inspect starting from this concept the program complexity measures so far, it will immediately strike us, that none of them takes into account neither the influence of the technique of **hiding** (e.g. use of procedures) on the complexity of programs nor the complexity of **data** used in the program, respectively the complexity of references to objects of different types or the consequences of hiding the representation and implementation of abstract data types. Parallelly with the control structures we also have to examine the structure of data with the help of an appropriate measure to their complexity.

According our original proposal [10], that when counting the complexity of a program, we should take the complexity of the data used and the complexity of data handling into consideration, we should see the decreasing of complexity through hiding techniques.

2 Definitions and notions

We shall define the new measure on the basis of the definitions given to the complexity of nested control structures. The definitons connected to this come from the excellently "rigorous" description of J.Howatt and A. Baker [5].

Definition 2.1. A *direct graph* $G = (N, E)$ consists of a set of nodes N and a set of edges E. An edge is an ordered pair of nodes (x,y). If (x,y) is an edge then node x is an *immediate predecessor* of node y and y is an *immediate successor* of node x. The set of all immediate predecessors of a node y is denoted **IP(y)** and the set of all immediate successors of a node x is denoted **IS(x)**. A node has *indegree n* if E contains exactly n edges of the form (w,z), similarly a node has *outdegree m* if E contains exactly m edges of the form (z,w).

Definition 2.2. A *path* P in a directed graph $G = (N, E)$ is a sequence of edges $(x_1, x_2), (x_2, x_3), \ldots (x_{k-2}, x_{k-1}), (x_{k-1}, x_k)$, where $\forall i[1 \leq i < k] \Rightarrow (x_i, x_{i+1}) \in E$. In this case P is a *path* from x_1 to x_k.

Definition 2.3. A *flowgraph* $G = (N, E, s, t)$ is a directed graph with a finite, nonempty set of nodes N, a finite, nonempty set of edges E, $s \in N$ is the start node, $t \in N$ is the terminal node. For any flowgraph G, the s start node is the unique node with indegree zero; the t terminal node is the unique node with outdegree zero, and each node $x \in N$ lies on some path in G from s to t. Let N' denote the set $N - \{s, t\}$.

J.W. Howatt and A.L.Baker define the notion of the basic block for modeling control flow as follows:

Definition 2.4. A *basic block* is a sequential block of code with maximal length, where a sequential block of code in a source program P is a sequence of tokens in P that is executed strating only with the first token in the sequence, all

the tokens in the sequence are always executed sequentially, and the sequence is always exited at the end. Namely, it doesn't contain any loops or if statements.

Definition 2.5. Every node $n \in N$ of a flowgraph $G = (N, E, s, t)$ which has outdegree greater than one is a *predicate node*. Let Q denote the set of predicate nodes in G.

The well-known measure of McCabe (cyclomatic complexity) is based only on the number of predicates in a program:$V(G) = p + 1$. The inadequacy of the measure becomes clear, if we realize that the complexity depends basically on the nesting level of the predicate nodes. The measures proposed by Harrison and Magel [3],[4] and Piwowarski [8] proven to be equivalent in principle by Howatt and Baker [5] take this lack into account.

Definition 2.6. Given a flowgraph $G = (N, E, s, t)$, and $p, q \in N$, node p *dominates* node q in G if p lies on every path from s to q. Node p *properly dominates* node q in G if p dominates g and $p \neq q$. Let $r \in N$, node p is the *immediate dominator* of node q if (i) p properly dominates q and (ii)if r properly dominates q then r dominates p.

The formal definition of the scope number is based on the work of Harrison and Magel.

Definition 2.7. Given a flowgraph $G = (N, E, s, t)$, and $p, q \in N,$, the set of *first occurence paths* from p to q, FOP(p,q) is the set of all paths from p to q such that node q occurs exactly once on each path.

Definition 2.8. Given a flowgraph $G = (N, E, s, t)$, and nodes $p, q \in N,$, the set of nodes that are on any path in FOP(p,q) is denoted by MP(p,q):

$$MP(p, q) = \{v \mid \exists P \; [\; P \in FOP(p.q) \wedge v \in P \;] \;\}$$

Definition 2.9. In a flowgraph $G = (N, E, s, t)$, the set of *lower bounds* of a predicate node $p \in N$ is

$$LB(p) = \{v \mid \forall r \forall P \; [\; r \in IS(p) \wedge P \in FOP(r, t) \;\Rightarrow\; v \in P \;]\}$$

Definition 2.10. Given a flowgraph $G = (N, E, s, t)$, and a predicate node $p \in N$, the *greatest lower bound* of p in G is

$$GLB(p) = \{q \mid \; q \in LB(p) \wedge \forall r \; [\; r \in (LB(p) \setminus \{q\}) \Rightarrow r \in LB(q)]\}$$

Definition 2.11. Given a flowgraph $G = (N, E, s, t)$, and a predicate node $p \in N$, the set of nodes predicated by node p is

$$Scope(p) = \{n \mid \exists q \; [\; q \in IS(p) \wedge n \in MP(q, GLB(p)) \;] \;\} \setminus \{\; GLB(p) \;\}$$

Definition 2.12. Given a flowgraph $G = (N, E, s, t)$, the set of nodes that *predicate* a node $x \in N$, is

$$Pred(x) = \{p \mid \; x \in Scope(p)\}$$

.

Definition 2.13. The *nesting depth* of a node $x \in N$, in a flowgraph $G = (N, E, s, t)$ is

$$nd(x) = \mid Pred(x) \mid$$

Thus, the total nesting depth of a flowgraph G was counted as

$$ND(G) = \sum_{n \in N'} nd(n)$$

The measure of program complexity given by Harrison and Magel is the sum of the adjusted complexity values of the nodes. This value can be given - as proved by Howatt - as the scope number of a flowgraph:

Definition 2.14. The *scope number*, SN(G) of a flowgraph $G = (N, E, s, t)$ is

$$SN(G) = \mid N' \mid + ND(G)$$

The main concept behind this definition is, that the complexity of understanding a node depends on its nesting depth, on the number of predicates dominating it. This measure was proved by J.W. Howatt and A.L.Baker to be equivalent to the ones proposed by Piwowarski or Dunsmore and Gannon, that is why we shall refer to this in the following.

3 The complexity of procedural programs

As we can see from the above, the software complexity measures did not so far take the role of procedures into consideration, while the complexity of data used was completely out of the question.

Our first suggestion is directed towards the introduction of the notion of **procedure** . The complexity of programs, decomposed to suitable procedures, is decreasing. We need a measure which expresses this observation.

Let us represent a program consisting of procedures not with a flowgraph, but with the help of a **set of flowgraphs**. Let us define the complexity of a program as the sum of the complexities of its component flowgraphs!

Definition 3.1. A *programgraph* $\mathcal{P} = \{G \mid G = (N, E, s, t)\ flowgraph\}$ is a set of flowgraphs, in which each start node is labelled with the name of the flowgraph. These labels are unique. There is a marked flowgraph in the set, called the 'main' flowgraph, and there is at least one flowgraph in the set which contains a reference to each label except the 'main' one.

Definition 3.2. The *complexity of a programgraph* will be measured by the sum of the scope numbers of its subgraphs

$$\mathcal{C}(\mathcal{P}) = \sum_{G \in \mathcal{P}} SN(G)$$

This definition shall reflect properly our experience that if we e.g. take a component out of the graph which does **not** contain a predicate node to form a procedure - i.e. a basic block, or a part of it (this means a single node), then we increase the complexity of the whole program according to our definition.

We extend the definitions that we have used so far: Let the set of nodes of our flowgraphs be widened by a new kind of node to denote the data! Let us denote by a small triangle ($\triangle$) the data nodes in the program. Let us draw to these nodes special edges, called **data reference edge**, which surely return to their origin from each node, where there is a reference to that data.

Definition 3.3. Let N and D be two finite, nonempty sets of control structure and data nodes respectively. A **data reference edge** is a triple (x_1,x_2,x_1) where $x_1 \in N$ and $x_2 \in D$.

Let us redefine the notion of a flowgraph as follows:

Definition 3.4. A *data-flowgraph* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ is a directed graph with a finite, nonempty set of nodes $\mathcal{N} = N \bigcup D$, where N represents the nodes belonging to the control structure of the program and D represents the nodes belonging to the data used in the program, (both of them are nonempty), with a finite, nonempty set of edges $\mathcal{E} = E \bigcup R$, where E represents the edges belonging to the control structure of the program, and R represents the set of its **data reference edges**. $s \in N$ is the start node, $t \in N$ is the terminal node. The s start node is always the unique node with indegree zero for all the data-flowgraphs $\mathcal{G}$,; the t terminal node is the unique node with outdegree zero, and each node $x \in \mathcal{N}$ lies on some path in $\mathcal{G}$ from s to t. Let $\mathcal{N}'$ denote the set $\mathcal{N} - \{s, t\}$.

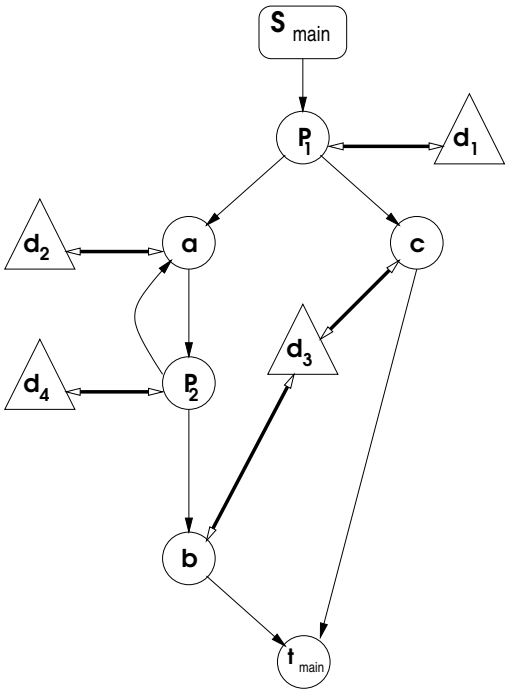

Fig.1.

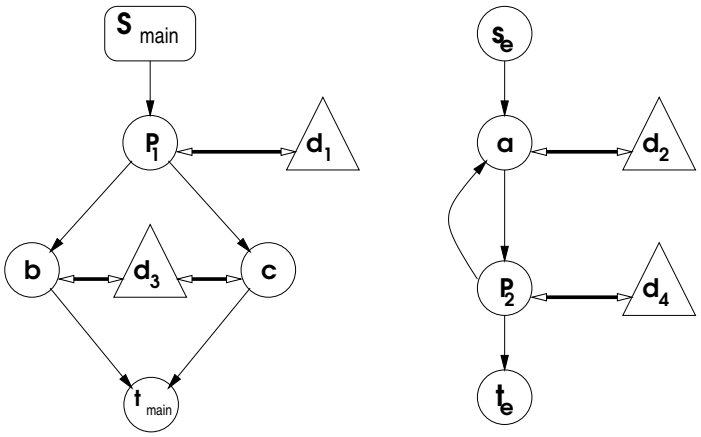

Fig.2.

As an example see Fig. 2. constructed from the graph shown on Fig.1.. The complexity of this program will be 18 opposed to the value 24 obtained for the program on Fig. 1.

The occurences are of course, in general, not so simple because there can be several references to the same data. How could we decrease the complexity of the program in addition to this? One fundamental tool is the decreasing of the number of small triangles, the number of data used. One possibility for this is that we draw certain data into one structure, creating data structures from our data. E.g. if we have to work with a complex number, then we decrease the complexity of the program if instead of storing its real and imaginary part separetely in the variables a and b , we draw these to a complex number x which has operations treating the real and imaginary part appropriately. The reduction(the decreasing of data nodes) occurs of course only when we **hide the components** in the following from the outerworld, since if we do not do this, this would mean, on the level of the program graph, that we did not merge two data nodes into one, but created a third one to the previous two.

As a matter of fact we can decrease the complexity of program in connection with data if and only if we build abstract data types **hiding the representation**. In this case the references to data elements will always be references to data since a data can only be handled through its operations. While computing the complexity of the whole program we have to take into account not only the decreasing of the complexity, but also its increase by the added complexity of determined by the implementation of the abstract data type. Nevertheless this will only be an additiv factor instead of the previous nested factor.

That is the most important complexity-decreasing consequence of the object oriented view of programming: **the object hides the type** from the predicates (decisions) supervising the use of the object.

4 The complexity of class

We can naturally extend our model to object-oriented programs. In the centre of the object-oriented paradigm there is the class. Therefore we should first describe how we measure the complexity of a class. In the base of the previous sections we can see the class definition as a set of (local) data and a set of methods.

Definition 4.1. An *class-graph* $\mathcal{O} = \{G \mid\ G\ data-flowgraph\}$ is a finite set of data-flowgraphs (the **member graphs**). The set of nodes $\mathcal{N} = N \bigcup D$, where N represents the nodes belonging the control structure of one of the member graphs and D represents the data nodes used by the member graphs. We can call D also as the **set of attributes** of the class. The set of edges $\mathcal{E} = E \bigcup R$ represents the E edges belonging the control structure of one of the member graphs and R as the data reference edges of the attributes. As the control nodes (nodes belonging to the control structure of one of the member graphs) were unique, there is no path from one member graph to another one. However, there could be attributes (data nodes) which are used by more than one member graph. These attributes have data reference edges to different member graphs.

This is a natural model of the class. It reflects the fact, that a class is a coherent set of attributes (data) and the methods working on the attributes. Here the **methods** (member functions) are procedures represented by individual data-flowgraphs (the member graphs). Every member graph has his own start node and terminal node, as they are individually callable functions. What does make this set of procedures more than an ordinary library is the *common set of attributes* used by the member procedures. Here the attributes are not local to one procedure but local to the object, and can be accessed by more procedures. (In the same time it is still possible to use local data to one procedure. This is a data node, which has reference edges to only one procedure.)

Definition 4.2. The *complexity of a class* can be computed in the very similar way than the complexity of the program:

$$\mathcal{C}(\mathcal{O}) = \mid N' \mid + \sum_{G \in \mathcal{O}} ND(\mathcal{G})$$

The complexity is depending on the nodes (both attributes and the ones involved in the control structure of a member) and the predicates dominating them. As in the definition of the complexity of a program, we summarise the total nesting depth of each method, and add the number of nodes of the object (both control and data nodes). When we define the $Pred(x)$ set of an attribute x, we involve the nodes from all the member graphs which predicate x. The number of control nodes in the object-graph is the sum of the number of control nodes of the member graphs. At the same time each attribute node is counted only once.

The definition reflects the common experience that good object-oriented programs have very strong binding between the attributes and the methods inside the class and have very week connection between different classes. The complexity of an individual member function doesn't depend on whether it works on global data or in a class attribute. However, the whole program can decrease its complexity if it uses member functions for access local data hidden in a class.

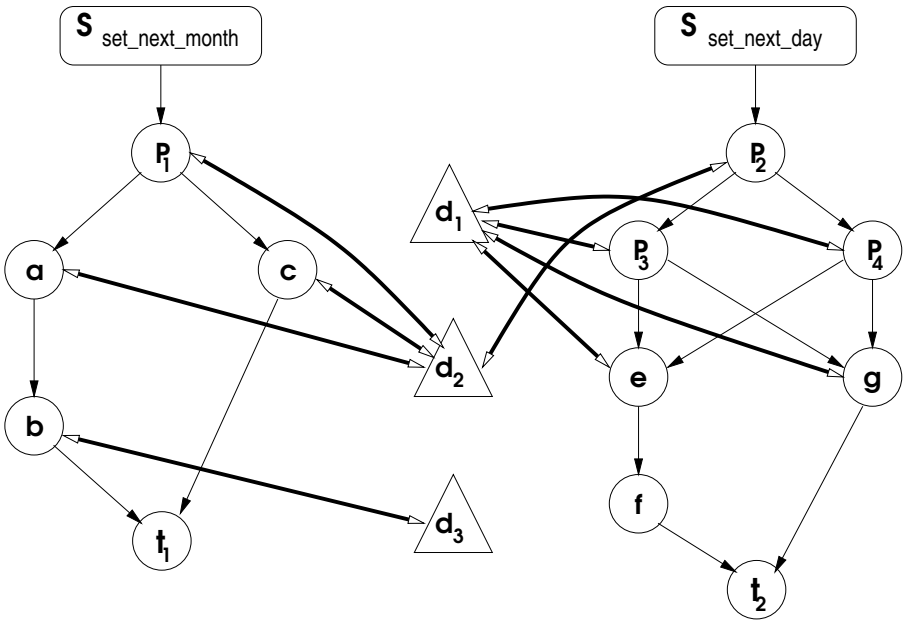

Fig.3.

See Fig. 5. as an example. Here we show a (simplified) *date* class. The class is represented by three data members: *day*, *month* and *year* denoted by d_1, d_2 and d_3 on Fig. 5. We implemented two methods on the class: *set_next_month* and *set_next_day*. We coded the *predicate* nodes as

p_1 ::= $month == 12$
p_2 ::= $month == 1$ *or* $month == 3$ *or ... or* $month == 12$
p_3 ::= $day == 31$
p_4 ::= $day == 30$

and the other nodes as

a ::= $month := 1$
b ::= $year := year + 1$
c ::= $month := month + 1$
e ::= $day := 1$
f ::= set_next_month
g ::= $day := day + 1$

(For the sake of simplicity we ignored february and the leap-year problem.) The complexity of the class can be computed based on the definition 4.2.
$Scope(p_1) = \{d_2, a, b, d_3, c\}$
$Scope(a) = \{d_2\}$
$Scope(b) = \{d_3\}$
$Scope(c) = \{d_2\}$
$Scope(p_2) = \{d_2, p_3, d_1, e, f, p_4, g\}$

$Scope(p_3) = \{d_1, e, f, g\}$
$Scope(p_4) = \{d_1, e, f, g\}$
$Scope(e) = \{d_1\}$
$Scope(f) = 0$
$Scope(g) = \{d_1\}$
$Pred(p_1) = 0$
$Pred(a) = \{p_1\}$
$Pred(b) = \{p_1\}$
$Pred(c) = \{p_1\}$
$Pred(p_2) = 0$
$Pred(p_3) = \{p_2\}$
$Pred(p_4) = \{p_2\}$
$Pred(e) = \{p_2, p_3, p_4\}$
$Pred(f) = \{p_2, p_3, p_4\}$
$Pred(g) = \{p_2, p_3, p_4\}$
$Pred(d_1) = \{p_2, p_3, p_4, e, g\}$
$Pred(d_2) = \{p_1, a, c\}$
$Pred(d_3) = \{p_1, b\}$
Thus $\mid N' \mid = 13$, $ND(\mathcal{G}) = 24$ and $\mathcal{C}(\mathcal{O}) = 37$.

Remarks

(1) In a real object-oriented language when we implement a class, member functions often call another member functions (from the same or from a different class). In Fig. 5. member function *set_next_day* calls *set_next_month*, an other member function of the same class. As we have seen at the procedural programs, we can represent the called procedure in the caller as an elementary node. The called procedure should represented as a member graph in the same or in a different object. As at the complexity of the program, such constructions are regularly decrease the complexity of the caller function and the class.

(2) It is possible, that a class has only data and has no member function. This is a kind of grouping data. However this is not a supported technique, it is syntactically correct to write in C++:

```
class date // example 4.1.
{
public :
    int year, month, day;
};
```

In our model, we can still compute the complexity of this class, which is the number of the attributes. This is in harmony with the common sense. Similarly, if a class has no attributes at all, its complexity is the sum of the individual members. This "class" is an ordinary *function library*.

(3) The model we introduced does not make difference between *public* and *private* data and method. The access right of a member is not influences the complexity of the class itself. Let see the two C++ class definitions bellow:

```
class date // example 4.2.
{
public :
    void set_next_month() {
        if(month == 12) { month = 1; year = year + 1; }
        else { month = month + 1; }
    }
    int year, month, day;
};
class date // example 4.3.
{
public :
    void set_next_month() {
        if(month == 12) { month = 1; year = year + 1; }
        else { month = month + 1; }
    }
private :
    int year, month, day;
};
```

Can an ordinary C++ programmer see the differences *in complexity* between the two definitions? We can hardly say yes. However, there could be differences in the complexity of the *client code*, which uses the class. If the client accesses the attributes of the class via the *set_next_month* function, we can replace its subgraph in the client code in the known way. This decreases the complexity of the client code. If the client accesses the attributes directly, we cannot do this. Using the example 4.2. the client programmer can choose whichever way, with the code in example 4.3. he is enforced to choose the better one.

5 Conclusions

The complexity measure studied here expresses the structural complexity of the program.

We investigated the given complexity measures, and found them suffering from a common problem, that they, while computing the complexity of a given program, did not take the role of neither the modularization nor the data used into account. On the basis of the previous efforts of J.W.Howatt and A.L.Baker we suggested a new measure of program complexity, which reflects our psychological feeling that the main concepts of object-oriented programming methodology help us to decrease the total complexity of a program.

The notion of inheritance allows actually to **hide a class of types**, further decreasing the sum of complexity, of course adding the complexity of the inheritance graph. To compute the complexity of an inheritance graph we have to use the graphrepresentation suggested by Meyer [7], namely using edges from the descendants to their ancestors, since the complexity of a class depends on their ancestor(s), not on their descendant(s). The complexity of an object-oriented

program will thus be determined by the sum of the complexity of the inheritance graph and the complexity of classes used.

References

[1] Dijkstra,E.W.: A Discipline of Programming, Prentice-Hall, Engelewood Cliffs, N.Y.,1976.

[2] Fóthi,Á. and Nyéky-Gaizler,J. : A Theoretical Approach of Objects and Types, in: Kai Koskimies and Kari-Jouko Raiha (eds.): Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala, Finland, August 21-23,1991, Report A-1991-5,August,1991.

[3] Harrison,W.A. and Magel,K.I. : A Complexity Measure Based on Nesting Level, ACM Sigplan Notices,16(3),63-74 (1981).

[4] Harrison,W.A. and Magel,K.I. : A Topological Analysis of the Complexity of Computer Programs with Less Than Three Binary Branches, ACM Sigplan Notices,16(4), 51-63 (1981).

[5] Howatt,J.W. and Baker,A.L. : Rigorous Definition and Analysis of Program Complexity Measures : An Example Using Nesting, The Journal of Systems and Sofware 10,139-150 (1989).

[6] McCabe, T.J. A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4),308-320 (1976).

[7] Meyer,B. : Object-Oriented Software Construction, Prentice Hall, New York, 1988

[8] Piwowarski,P. : A Nesting Level Complexity Measure, ACM Sigplan Notices ,17(9),44-50 (1982).

[9] Varga, L.: A new approach to defining software design complexity. In: R.Mittermeier (ed.): Shifting Paradigms in Software Engineering. Springer Verlag, Wien, New York, 198-204.(1992)

[10] Fóthi, Á., Nyéky-Gaizler, J.: On the Complexity of Object-Oriented Programs, Proc. of the Third Symp. on Programming Languages and Software Tools, Kaariku, Estonia, 1993.