PERGAMON Mathematical and Computer Modelling 0 (2003) 1–0
www.elsevier.com/locate/mcm

The Structured Complexity of Object-Oriented Programs

Á. Fóthi, J. Nyéky-Gaizler and Z. Porkoláb
Department of General Computer Science
Eötvös Loránd University
Pázmány Péter sétány 1/D., H-1117 Budapest, Hungary
<fa><nyeky><gsd>@ludens.elte.hu

Abstract—There are several methods measuring the complexity of object-oriented programs. Most of them are based on some special object-oriented feature: number of methods/classes, cohesion of classes, inheritance, etc. In practice, however, object-oriented programs are constructed with the help of the same control structures as traditional ones. Moreover, recent ideas of multiparadigm programming (i.e., emerging use of generic programming and aspect-oriented programming) has the effect that in modern programs—and even in class libraries—object-orientation is only one (however major) construction tool among others. An adequate measure therefore should not be based on special features of *one* paradigm, but on basic language elements and construction rules which could be applied to many different paradigms.

In our model discussed here, the complexity of a program is the sum of three components: the complexity of its control structure, the complexity of data types used, and the complexity of the data handling (i.e., the complexity of the connection between the control structure and the data types). We suggest a new complexity measure. First, we show that this measure works well on procedural programs, and then we extend it to object-oriented programs.

There is a software tool under development based on gnu g++ compiler which computes our new measure. We can apply this tool to C and C++ sources to gain a number of quantitative results with our measure.

Keywords—Software metrics, Object-oriented programming, Complexity.

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) and on the number of predicates (cyclomatic complexity). However, 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 (procedural) programs was so far mostly measured either on the basis of its control structure or on the data they used [1].

Nowadays, one of the most frequently used "words" 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 simpler 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

0895-7177/03/$ - see front matter Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
PII:00

programs. Followers of object-oriented methodology state that professional software production becomes notably simplified using this new technique, which results in enormous cost decrease. Measuring object-oriented software is often based on special object-oriented notions.

However, modern programming languages implement facilities not only for the object-oriented paradigm. These languages still have very strong procedural parts. Also, new emerging techniques, most cases orthogonal to object orientation, are used in cooperation with object theory. The C++ language is one of the most frequently used modern languages in Europe—uses *generic programming* techniques: *templates*, *traits*, *iterators* with cooperation of classes, inheritance, etc. [2]. SUN plans to invent templates for the Java language, too. *Aspect-oriented* programming [3] is another important technique from XEROX Palo Alto Research Center. Aspects are applied to object-oriented Java programs to reduce complexity, eliminating cross-cutting code segments. These paradigms are heavily mixed in recent programs [4,5]. Inheritance from a template class, generic algorithms receiving template parameters are everyday practices in C++, including the standard library. It seems illogical to measure such programs or libraries only from the point of one paradigm.

An adequate measure, therefore, should not be based on special features of *one* paradigm, but on basic language elements and construction rules applied to different paradigms. Based on the previous works of Piwowarski [6], Harrison and Magel [7,8], and Howatt and Baker [9], we introduce a new measure of complexity.

Our main proposal is 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. Accordingly, the complexity of a program depends on three components:

(i) the complexity of its control structure,
(ii) the complexity of data types used, and
(iii) the complexity of the data handling (i.e., the complexity of the connection between the control structure and the data types).

We suggest a new measure of complexity of a program or (class) library. First, we show this measure working well on procedural programs. Then, we extend the measure to object-oriented programs in the following steps.

(1) We define the complexity of class. Class is defined as a set of data (attributes) and control structures (member functions, methods) working on the attributes. We define the complexity of class as a sum of the complexity of attributes and the complexity of member functions. The definition reflects the common experience that good object-oriented programs have very strong bindings between the attributes and the methods inside the class and have weak connections between different classes. The measure is also examined on special cases, such as member functions calling other member functions, and classes with no attributes (old-style libraries), etc.
(2) We examine the complexity issues of the connection of classes. Inheritance and aggregate relationships between classes can increase global complexity of the program. However, every time we use the same class we can see the benefit of these constructions. In some classical examples, we show how complexity depends on binding between classes.

2. PRELIMINARY DEFINITIONS AND NOTIONS

We shall define the new measure on the basis of the definitions given to the complexity of nested control structures. The definitions connected to this come from the "rigorous" description of [9].

DEFINITION 1. *A directed 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 $\mathbf{IP(y)}$ *and the set of all immediate successors of a node x is denoted* $\mathbf{IS(x)}$. *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. *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 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 $N - \{s, t\}$.*

Howatt and Baker define the notion of the basic block for modeling control flow as follows.

DEFINITION 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 starting 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 does not contain any loops or if statements.*

DEFINITION 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 in [6–8], proven to be equivalent in principle (see [9]), take this lack into account.

DEFINITION 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 q 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 7. *Given a flowgraph $G = (N, E, s, t)$, and $p, q \in N$, the set of first occurrence paths from p to q,* $\mathrm{FOP}(p, q)$ *is the set of all paths from p to q such that node q occurs exactly once on each path.*

DEFINITION 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* $\mathrm{FOP}(p, q)$ *is denoted by $MP(p, q)$,*

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

DEFINITION 9. *In a flowgraph $G = (N, E, s, t)$, the set of lower bounds of a predicate node $p \in N$ is*

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

DEFINITION 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*

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

DEFINITION 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*

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

DEFINITION 12. *Given a flowgraph $G = (N, E, s, t)$, the set of nodes that predicate a node $x \in N$ is*

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

DEFINITION 13. *The nesting depth of a node $x \in N$, in a flowgraph $G = (N, E, s, t)$, is*

$$\mathrm{nd}(x) = |\mathrm{Pred}(x)|.$$

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

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

The measure of program complexity given by Harrison and Magel [7] is the sum of the adjusted complexity values of the nodes. This value can be given (as proved by Howatt [9]) as the scope number of a flowgraph.

DEFINITION 14. *The scope number*, $\mathrm{SN}(G)$, *of a flowgraph $G = (N, E, s, t)$ is*

$$\mathrm{SN}(G) = |N'| + \mathrm{ND}(G).$$

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 Howatt and Baker [9] to be equivalent to the ones proposed by Piwowarski [6] or Dunsmore and Gannon [10].

3. PROPOSAL FOR A NEW MEASURE

As we can see from Section 2, the software complexity measures did not yet 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 15. *A programgraph $\mathcal{P} = \{G \mid G = (N, E, s, t)$ flowgraph$\}$ is a set of flowgraphs, in which each start node is labeled with the name of the flowgraph. These labels are unique. There is a marked flowgraph in the set, called the main flowgraph, and to each label except the main one there is at least one flowgraph in the set which contains a reference to its start node.*

DEFINITION 16. *The complexity of a programgraph will be measured by the sum of the scope numbers of its subgraphs*

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

This definition shall reflect properly our experience. For example, if we 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. This is a direct consequence of the fact that, in our measures so far, we contracted the statement sequences that are reasonable according to this view of complexity. If we create procedures from sequences, the program becomes more difficult to follow. Since we cannot read the program linearly, we have to "jump" from the procedures back and forth. The reason for this is that a sequence of statements can always be viewed as a single transformation. This could, of course, be refined by counting the different transformations as being of different weight, but this approach would transgress the competence of the model used. The model mirrors these considerations since if we form a procedure from a subgraph containing no predicate nodes, then the complexity increases according to the complexity of the new procedure subgraph (i.e., by 1).

On the other hand, if the procedure does contain predicate node(s), then, by modularization, we decrease the complexity of the whole program depending on the nesting level of the outlifted procedure. If we take a procedure out of the flowgraph, creating a new subgraph out of it, the measure of its complexity becomes independent of its nesting level. On the place of the call, we may consider it as an elementary statement (as a basic block, or part of it).

See Figures 1 and 2 as an example. It is evident that even in such a simple case the complexity of the whole program decreases if we take an embedded part of the program out as a procedure. One can readily see the complexity of the program shown on Figure 1 $\mathrm{SN}(G) = 19$, while the complexity of the second version shown on Figure 2 $\mathcal{C}(\mathcal{P}) = \sum_{G\in\mathcal{P}} \mathrm{SN}(G) = 18$.

This model reflects well the experience of programmers that the complexity of a program can be decreased by the help of modularization not only when the procedure is called from several points of the program, but a well-developed procedure alone, in the case of a single call, can also decrease the complexity of the whole program. Certainly, when a procedure is called more than once, the decrease of complexity is greater.

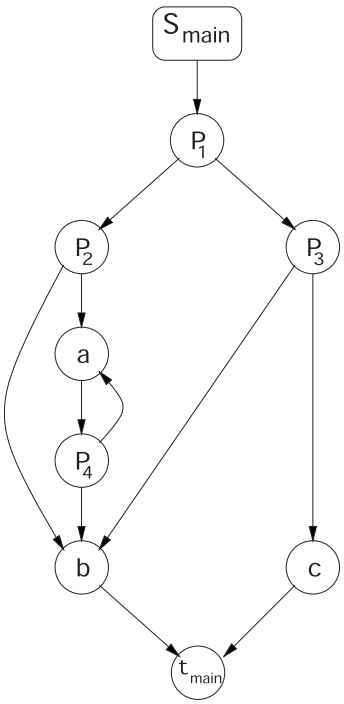


Figure 1.

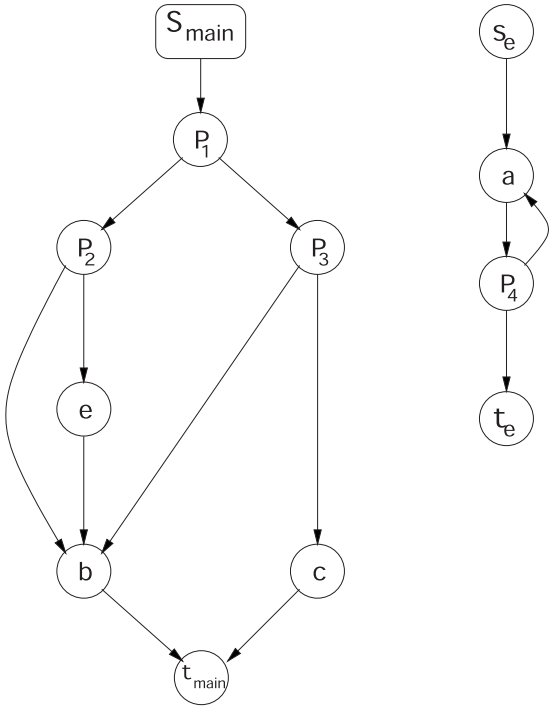


Figure 2.

It is also trivial that if we form a procedure from the whole program, then we also increase the complexity.

Now, we are reaching the point where it is inevitable, not only from the point of view of handling procedure calls but also in connection with the whole program, to deal with the question of data. The complexity of a program depends not only on the complexity of the transformation but also on the subject of this transformation—the data to be processed.

We extend the definitions that we have used so far. Let the set of nodes of our flowgraphs be extended by a new kind of node to denote the data. Let us denote by a small triangle (Δ) the data nodes in the program. Data nodes and control nodes are connected by special edges, called *data reference edges*. A data reference edge is directed; the direction reflects the flow of information between the data and the control node.

DEFINITION 17. *Let N and D be two finite, nonempty sets of control structures and data nodes, respectively. A data reference edge is a pair (x_1, x_2) where either $x_1 \in N$ and $x_2 \in D$ or $x_1 \in D$ and $x_2 \in N$.*

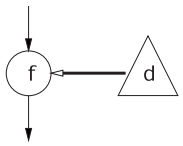
Let us redefine the notion of a flowgraph as follows.

DEFINITION 18. *A data flowgraph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ is a directed graph with a finite, nonempty set of nodes $\mathcal{N} = N \cup D$, 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 $\mathcal{E} = E \cup R$, 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 $\mathcal{N} - \{s, t\}$.*

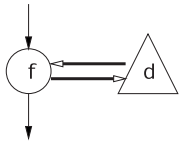
Here p_1 and p_2 predicate nodes read d_1 and d_4 data nodes. a writes d_2, b reads d_3, and c reads and writes d_3. The latter fact is reflected in the two edges between c and d_3, and therefore the $c - d_3$ connection should be taken into account twice. A very precise notation would replace d_3 with d_{3r} and d_{3w} nodes to distinguish the *read* and *write* operations. For the sake of notational simplicity, in this article we use the d_3 notation for both.

At the first moment, it seems unnecessary to distinguish between read and write access. However, to create a really fine-grain measure, this kind of difference is important. In the C++ example in Figure 3, we call function f with parameter d. The complexity of the call depends on the declaration of f.

```
int d;
// ...
f( d);
```




```
void  f( int d);
```




```
void  f( int &d);
```

Figure 3.

In the first case, the parameter is passed by value; the communication between d and the function goes one way. In the second case, the parameter is passed by reference, so the function can also modify d. Therefore, the second call is more complex than the first one—independently from the code of function f. This is also a good example for Weyuker's sixth axiom [11], code occurring after different but equally complex prologues (here, the different definition of parameter passing to f) may be differently affected by the distinct prologues.

The complexity of the program will be computed from the set of graphs obtained this way in accordance with the previous definitions depending on the number of nodes and predicates dominating them. We call attention to the fact that if we take the role of data in the program into consideration, then the number of those nodes which have an outdegree greater than one increases, and we have to determine the Scope also for those nodes, where there is a reference to data.

As an example, let us have a look at the program represented by the graph in Figure 4. The complexity counted this way can be obtained.

$\mathtt{Scope}(p_1) = \{d_1, a, d_2, p_2, d_4, c, d_3, d_3, b\}$
$\mathtt{Scope}(p_2) = \{a, d_2, p_2, d_4\}$
$\mathtt{Scope}(a) = \{d_2\}$
$\mathtt{Scope}(b) = \{d_3\}$
$\mathtt{Scope}(c) = \{d_3, d_3\}$
$\mathtt{Pred}(p_1) = 0$
$\mathtt{Pred}(p_2) = \{p_1, p_2\}$
$\mathtt{Pred}(a) = \{p_1, p_2\}$
$\mathtt{Pred}(b) = \{p_1\}$
$\mathtt{Pred}(c) = \{p_1\}$
$\mathtt{Pred}(d_1) = \{p_1\}$

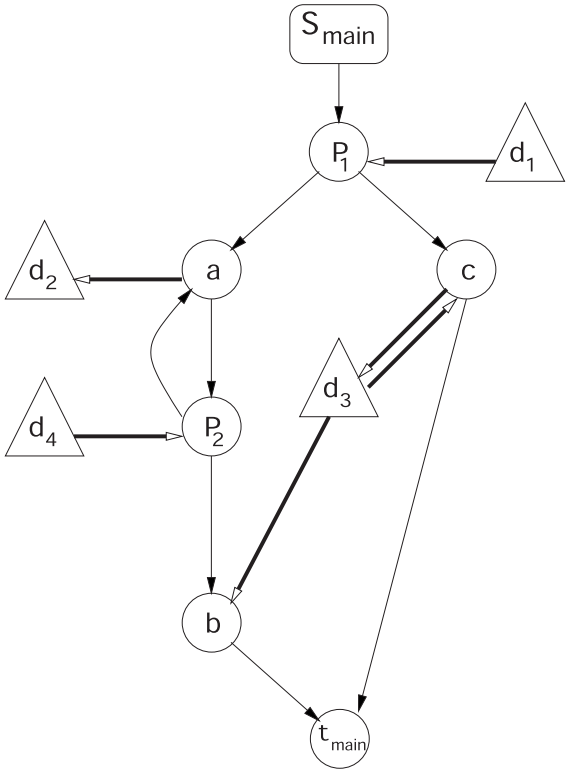


Figure 4.

$\texttt{Pred}(d_2) = \{p_1, p_2, a\}$
$\texttt{Pred}(d_3) = \{p_1, p_1, b, c, c\}$
$\texttt{Pred}(d_4) = \{p_1, p_2\}$

Thus, $\mathrm{ND}(\mathcal{G}) = 17$ and $\mathrm{SN}(\mathcal{G}) = 26$.

This way, the complexity will also be influenced by the data and the extent data makes a program more complicated is determined by the decisions preceding the reference to it. This graph and its complexity measure defined this way express that the complexity of a program depends also on the data used, and on the references to these data.

As we have seen so far, the complexity may be decreased by the appropriate modularization of the program. Similarly, if we take out a subgraph which contains one or more data with all of the data reference edges leading to this data, we will decrease the complexity. For example, if there is a single reference to a data at some transformation, and we take this transformation in order to create a procedure, where this data will be a *local* variable—the complexity of the program decreases. The substantial moment in this activity is that we *hide* a variable from the view of the whole program, we make it invisible (local), and thus, we essentially decrease the additive factor to the complexity at this point.

As an example, see Figure 5 constructed from the graph shown in Figure 4. The complexity of this program will be 23 opposed to the value 26 obtained for the program in Figure 4. The reduction of the complexity value is the result partly of hiding the predicate node p_1, and partly making data nodes d_1 and d_2 "local" to procedure e.

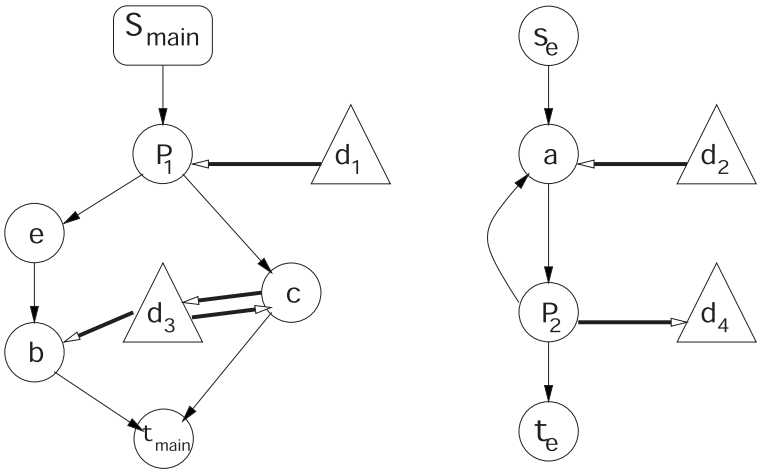


Figure 5.

The occurrences 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? 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. For example, when 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 separately 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 in addition to the previous two.

As a matter of fact, we can decrease the complexity of a program in connection with data if we build abstract data types *hiding the representation*. In this case, the references to data elements will always be references to data since 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 determined by the implementation of the abstract data type. Nevertheless, this will only be an additive factor instead of the previous nested factor.

That is the most important complexity-decreasing consequence of the object-oriented view of programming: *the class hides its representation* (both data structure and algorithm) from the predicates (decisions) supervising the use of the object of class.

4. THE COMPLEXITY OF CLASSES

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. On the basis of the previous sections, we can see the class definition as a set of (local) data and a set of methods.

DEFINITION 19. *A class graph $\mathcal{O} = \{G \mid G \text{ data flowgraph}\}$ is a finite set of data flowgraphs (the member graphs). The set of nodes $\mathcal{N} = N \cup D$, 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 also call D the set of attributes of the class. The set of edges $\mathcal{E} = E \cup R$ represents the E edges belonging to 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) are 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 makes 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 several procedures.

At the same time, it is still possible to use local data for one procedure. This is a data node, which has reference edges to only one procedure. In our model there is no complexity difference whether a data node is local to a method or it is a private attribute accessed by only that method.

DEFINITION 20. *The complexity of a class* can be computed in a very similar way to the complexity of the program,

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

The complexity is depending on both attributes and control nodes (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 sum the total nesting depth of each method, and add the number of nodes of the object (both data and control nodes). When we define the $\mathrm{Pred}(x)$ *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 bindings between the attributes and the methods inside the class and have very weak connections between different classes. The complexity of an individual member function does not 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 accessing local data hidden in a class. See Figure 6 as an example. Here, we show a (simplified) *date* class. The class is represented by three

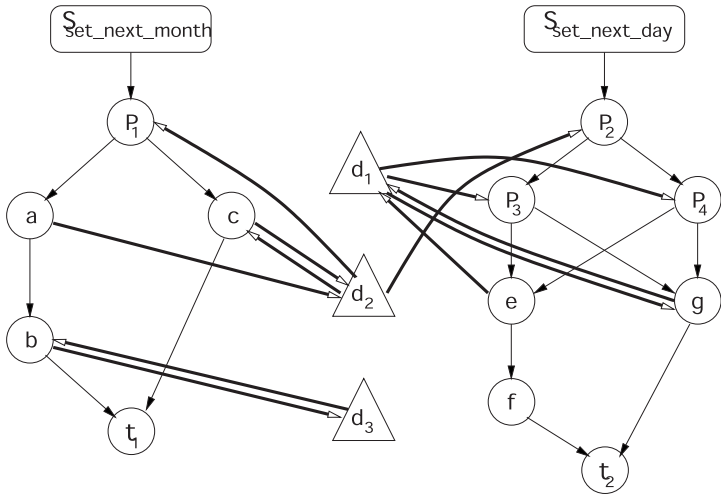


Figure 6.

data members: *day*, *month*, and *year* denoted by d_1, d_2, and d_3 on Figure 6. We implemented two methods on the class: *set_next_month* and *set_next_day*. We coded the *predicate* nodes as

```
p1 ::=  month == 12
p2 ::=  month == 1 or month == 3 or ...or month == 12
p3 ::=  day == 31
p4 ::=  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 Definition 20.

```
Scope(p1) = {d2,d2,a,b,d3,d3,c}
Scope(a) = {d2}
Scope(b) = {d3,d3}
Scope(c) = {d2,d2}
Scope(p2) = {d2,p3,d1,d1,e,f,p4,g}
Scope(p3) = {d1,d1,e,f,g}
Scope(p4) = {d1,d1,e,f,g}
Scope(e) = {d1}
Scope(f) = 0
Scope(g) = {d1,d1}
Pred(p1) = 0
Pred(a) = {p1}
Pred(b) = {p1}
Pred(c) = {p1}
Pred(p2) = 0
Pred(p3) = {p2}
Pred(p4) = {p2}
Pred(e) = {p2,p3,p4}
```

Pred(f) = {p_2,p_3,p_4}
Pred(g) = {p_2,p_3,p_4}
Pred(d_1)={p_2,p_2,p_3,p_3,p_4,p_4,e,g,g}
Pred(d_2)={p_1,p_1,p_2,a,c,c}
Pred(d_3)={p_1,p_1,b,b}

Thus, $|N'| = 13$, $\mathrm{ND}(\mathcal{G}) = 33$, and $\mathcal{C}(\mathcal{O}) = 46$.

REMARK 1. In a real object-oriented language when we implement a class, member functions often call another member function (from the same or from a different class). In Figure 6, member function `set_next_day` calls `set_next_month`, another member function of the same class. As we have seen in the procedural programs, we can represent the called procedure in the caller as an elementary node. The called procedure should be represented as a member graph in the same or in a different object. As with the complexity of the program, such constructions regularly decrease the complexity of the caller function and the class.

REMARK 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 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*.

REMARK 3. The model we introduced does not make a distinction between *public* and *private* data and method. The access right of a member does not influence the complexity of the class itself. Let us see the two C++ class definitions below.

```
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 Example 4.2, the client programmer can choose whichever way; with the code in Example 4.3, he is forced to choose the better one.

It is easy to see that our measure fulfills all the axioms proposed in [11]. However, it was shown by Cherniavsky and Smith [12] that Weyuker's properties do not guarantee the correctness of a measure. We still believe that those axioms are minimal requirements, and a good measure should comply with them.

5. CLASS RELATIONSHIPS

A data member of a class is marked with a single data node regardless of its internal complexity. If it represents a complex data type, its definition should be included in the program and its complexity is counted there. Up to the point where we handle this data as an atomic entity, its effect to the complexity of the handler code does not differ from the effect of the most simple (built-in) types. In a C++ program a correctly written `date` class implements the *assign* (=), the *less-than* (<), and the *increment* (++) operations. From the viewpoint of the code using class `date`, the internal implementation of `date` makes no difference in the following statements.

```
date x, y;
//...
if ( x  <  y ) x  =  y;
else ++x;
```

Here, we use the member functions of `date`, the calls of which have constant complexity regardless of its implementation. However, if we break the encapsulation of class `date` (i.e., we directly access its components), the data reference edges connect the handler code to the internal representation and increment its complexity. Once again, we stress this fact has to do with the private or public members only in an indirect way: as far as we use the methods to handle data, it does not matter whether the components are public or private. Of course, the compiler supports this strategy only when we made our components private.

Inheritance is handled in a similar way. Code of the derived class in most cases (but not necessary) refers to the methods and/or data members of the base class(es). These references (method calls or data accesses) are described in the very same way as we did in the case of procedural programs. The motivation here is again to derive complexity from the basic, paradigm-independent program elements.

```
class diary : public date
{
public:
  string event;
};
```

Here, we used inheritance to extend the interface of `date`. There is no communication between the basic class and the new attribute(s) and methods (not shown here). In this case, the increase of complexity is the complexity of the new attributes and methods of the derived class.

In the case where derived class has a stronger binding to the base class(es), we use the known notations. Base class attributes accessed by derived methods are expressed by data reference edges; base class methods called by derived methods are represented by their nodes in the control graph of the caller method.

It is easy to see that the less the data reference edge crosses the border of the classes, the lower the complexity number we achieve. We stress again; this is rather a result of the specially organized code and not the basis of our definition.

6. CONCLUSIONS

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

We investigated the complexity measures given in [6–9], and found them suffering from a common problem—that they, while computing the complexity of a given program, did not take the role of either the modularization or the data used into account. On the basis of the previous efforts of Howatt and Baker [9], 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 us actually to *hide a class internal representation*, further decreasing the sum of complexity, of course adding to the complexity of the inheritance hierarchy.

Our measure is based on *paradigm-independent* notions. This makes it possible to use this method in a mixed-paradigm environment—a growing claim in modern programming languages today. Well-designed, fully implemented class interfaces, strong bindings inside a class, and weak bindings between a class and its environment, as well as good inheritance hierarchy are still major components to decrease complexity, yet they are not the base points to define our measure but a result of a specially organized code.

There is a gnu g++ software tool under development implementing our new measure. We can apply this tool to C and C++ sources to gain a number of quantitative results with our measure.

REFERENCES

1. S. Henry and D. Kafura, Software structure metrics based on information flow, *IEEE Trans. Software Engineering* **7**, 510–518, (1981).
2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, (1997).
3. G. Kiczales, Aspect-oriented programming (AOP), *ACM Computing Surveys* **28** (4es), (1996).
4. J.O. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, (1998).
5. K. Czarnecki and U.W. Eisenecker, *Generative Programming*, Addison-Wesley, (2000).
6. P. Piwowarski, A nesting level complexity measure, *ACM Sigplan Notices* **17** (9), 44–50, (1982).
7. W.A. Harrison and K.I. Magel, A complexity measure based on nesting level, *ACM Sigplan Notices* **16** (3), 63–74, (1981).
8. W.A. Harrison and K.I. Magel, A topological analysis of the complexity of computer programs with less than three binary branches, *ACM Sigplan Notices* **16** (4), 51–63, (1981).
9. J.W. Howatt and A.L. Baker, Rigorous definition and analysis of program complexity measures: An example using nesting, *The Journal of Systems and Software* **10**, 139–150, (1989).
10. Dunsmore and Gannon.
11. E.J. Weyuker, Evaluating software complexity measures, *IEEE Trans. Software Engineering* **14**, 1357–1365, (1988).
12. J.C. Cherniavsky and C.H. Smith, On Weyuker's axioms for software complexity measures, *IEEE Trans. Software Engineering* **17**, 1357–1365, (1991).
13. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, (1976).
14. Á. Fóthi and J. Nyéky-Gaizler, A theoretical approach of objects and types, In *Proceedings of the Second Symposium on Programming Languages and Software Tools*, Pirkkala, Finland, August 21–23, 1991, (Edited by K. Koskimies and K.-J. Raiha), Report A-1991-5, (August 1991).
15. T.J. McCabe, A complexity measure, *IEEE Trans. Software Engineering* **2** (4), 308–320, (1976).
16. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, New York, (1988).
17. L. Varga, A new approach to defining software design complexity, In *Shifting Paradigms in Software Engineering*, (Edited by R. Mittermeier), pp. 198–204, Springer-Verlag, Wien, (1992).
18. J.S. Davis and R.J. LeBlanc, A study of the applicability of complexity measures, *IEEE Trans. Software Engineering* **14**, 1366–1372, (1988).
19. K.B. Lakshmanan, S. Jayaprakash and P.K. Sinha, Properties of control-flow complexity measures, *IEEE Trans. Software Engineering* **17**, 1289–1295, (1991).
20. J. Tian and M.V. Zelkowitz, Complexity measure evaluation and selection, *IEEE Trans. Software Engineering* **21**, 641–650, (1995).