Annales Univ. Sci. Budapest., Sect. Comp. 19 (2000) 75-92

ABSTRACTION STRATEGIES IN PRACTICE

Á. Fóthi, J. Nyéky-Gaizler and É. Harangozó
(Budapest, Hungary)

Abstract. When qualifying a program, one puts the emphasis more and more on its reliability. The abstract approach to problems may mean a significant help to achieve this reliability. In this paper we would like to present two possible abstraction strategies: the "data abstraction" and the "function abstraction" - through the solution of a concrete problem, the updating of sequential files.

I. Introduction

From the factors of software quality the reliability of programs has come to the centre of interest with the development of large systems. The aim is to write "correct" programs which exactly perform their tasks, as defined by the requirements of the specification, thereby enabling a given relationship to be treated by means of computers. This goal is of course more easily stated than achieved. Here the first and fundamental step is - as it was pointed out by Naur [4] - that the programmer should set up his own theory on the given problem. Everything else (the structure of the system, etc.) follows from this start. Many things can be taught of the skills and tricks of programming but nobody will become a good programmer unless he/she is able to integrate all available relevant knowledge in a creative way to build up his/her own theory to solve the problem in question. Studying various programming methods, Floyd [5] stated that - at least for the time being - a programming method as a system of rules which would mechanically lead to the correct solution of any problem is a mere illusion.

Nevertheless, programming methods are extremely important in the education of programmers. Skill in solving typical problems and the knowledge of possible ways of approaches are determinative for the quality of the programmers' work. In the present paper the authors wish to contribute to this training

Supported by the Hungarian National Science Research Foundation under grant OTKA 2045.

by presenting widely applicable programming methods. In the following we shall show on a concrete example how abstraction as a technique of thinking can manifest itself in the solution of programming tasks. Our approach is twofold: we shall

- analyse the level(s) and depth of abstraction and its effect on the specification of the problem and

- demonstrate the effect of abstraction types on the solution of the problem.

The solution of a programming task is essentially based on abstraction. The depth of this abstraction depends on how general the concept is that was created by the programmer for modelling the problem. One should find the appropriate level of abstraction which is still informative for the given task and a further abstraction would be too general. In specifying a problem by determining the components of the state space and the pre- and postconditions, the above factors are already taken into account. The final, correct program is then worked out by some chosen methods - e.g. by top-down iterative refinement, or by constructing it from available components bottom-up, or by combining these two techniques.

In the process of abstraction, adequate concept-formation is extremely important. One should try to generalize the problem, i.e. to find a class of problems containing it. The correct solution of this wider class of problems obviously provides a more efficient tool, since the wanted concrete solution - as a special case - can be deduced from the general one. The application of the obtained higher level concept allows more transparent treatment (both theoretically and practically) compared to the specification written for the problem in first approximation.

The concrete strategy of solution can also be influenced by altering the specification. This can be done in two ways: by data abstraction or by function abstraction.

The first method, "data abstraction", means that a new data type will be defined for the exact description of the problem. Then, instead of using the initial state space, the problem is described by means of the new data type after state space transformation, and the operations of the new data type will be performed by the help of the initial data type.

The alternative method, "function abstraction", consists of the definition of an appropriate function to describe the postcondition. The solution of the concrete problem will be given by the value - or a series of values - of this function.

The above principles and methods can best be illustrated by an example. The updating of sequential files is one of the basic programming tasks. In the remaining of the paper the application of the above principles for the solution

of a concrete problem of this type is presented. (Concepts and notations see in [2,3].)

II. The definition of updating

Let us consider a data file consisting of a sequence of records of the same type. We shall call this the "master" file:

$T = it(E)$

$t_0 : T$

Furthermore there is given a file, called "modifier" file, which is a sequence of transformations:

$M = it(F) \quad F = \{f \mid f : T \to T \text{ relation}\}$

$m : M$

The task is to update the master file by the help of the modifier file, i.e. to produce a new master file by performing the sequence of transformations given in the modifier file. Let us denote this transformation by $t = upd(t_0, m)$.

This description is still too general, for the actual evaluation of the function it needs to be specialized by further restrictions.

1. Updating by means of key-values

Let t_0 and m be sequential files with key-values. What does it mean?

$T = seq(E)$ where $E = (K, D)$

$M = seq(F)$ where $F = (K, V)$,

where K is an arbitrary ordered set, the key part of the record, D is the type of the data component and V specifies the transformation to be performed. Let us assume that both files are ordered in nondecreasing way with respect to key-values.

2. Let t_0 be a single-valued file with respect to key-values, i.e. let it contain any key-value only once:

$$t_0 : T \quad \forall i, j \in [1, t_0.dom] : t_i.K = t_j.K \iff i = j$$

3. Let us introduce restrictions for the individual:
transformations:

$V = (W1; W2; W3)$, ";" denotes the union-type, where

$W1 = \{\text{deletion}\}$

$W2 = (\{\text{insertion}\}, D)$
$W3 = (\{\text{modification}\}, M')$
$M' = seq(G)$, where $g \in G$ if $g : D \to D$

Let us define the effect of these modifier transformations on the master file (we shall give the definiton for sets - this is sufficient because of the single-valuedness of t_0 with respect to key).

Let us define for an arbitrary $m_i \in M$:
the effect of the transformation "deletion" (case $m_i.W1$):

$$\{m_i(t)\} = \{t_j | t_j \in t \wedge t_j.K \neq m_i.K\}$$

The effect of the transformation "insertion" (case $m_i.W2$):

$$\{m_i(t)\} = \{t\} \cup \begin{cases} \emptyset, & \text{if } \exists k \in [1, t.dom] : t_k.K = m_i.K \\ \{(m_i.K, m_i.D)\}, & \text{if } \nexists k \in [1, t.dom] : t_k.K = m_i.K \end{cases}$$

The effect of the transformation "modification" (case $m_i.W3$):

$$\{m_i(t)\} = \{t_j | t_j \in t \wedge t_j.K \neq m_i.K\} \cup \begin{cases} \emptyset, & \text{if } \nexists k \in [1, t.dom] : t_k.K = m_i.K \\ H, & \text{if } \exists k \in [1, t.dom] : t_k.K = m_i.K \end{cases}$$

where $H = \{(m_i.K, m_i.M'(t_k.D)) \mid t_k \in t \wedge m_i.K = t_k.K\}$

Definition. An updating is called ordinary if it satisfies restrictions 1, 2 and 3.

Definiton. An ordinary updating is called simple if G is a single-element and constant transformation. Then a correction is a single constant transformation, in fact a change.

III. The case of the single-valued modifier file

Definition. A file is called deterministic, if it is single-valued with respect to key. If the key-values are not unique, the file is called nondeterministic.

Let us assume that the modifier file is also deterministic.

The specification of the problem:
The state space:

A : $T \times M \times T$
$t_0 \quad m \quad t$

B : $T \times M$
$t'_0 \quad m'$
The precondition:
Q : ($t_0 = t'_0 \wedge m = m'$ and update is simple and m' is deterministic)
The postcondition:
R : ($t = upd(t'_0, m')$)

III.1. Deduction of the problem to the union of sets

What key-values will be contained by t, the new master file? Certainly only the values contained by t'_0 or m' or both. Let us try to deduce the problem to the union of two sets! Let $K(t_0)$, and $K(m)$ denote the sets of key-values in t_0 and m, respectively, and $k(t_0)$ and $k(m)$ the elements in t_0 and m, respectively, the key of which is k.

Let us define an extension of the set of items: $D' = (D \cup \{"empty"\})$.

Both the domains of definition and the ranges of the individual transformations can be given by means of D':

$W1 : D \rightarrow \{"empty"\}$ - deletion
$W2 : \{"empty"\} \rightarrow D$ - insertion
$W3 : D \rightarrow D$ - modification

If the data part of an element turns into "empty", then this element actually does not exist and will not be contained by the new master file.

Remember the program $z = x \cup y$:

program:
$z := \emptyset$
while $x \neq \emptyset \vee y \neq \emptyset$ **loop**
$e :\in (x \cup y)$
case when
$e \in x \wedge e \notin y \rightarrow z := z \tilde{\cup} \{e\}$
$x := x \simeq \{e\}$
$e \notin x \wedge e \in y \rightarrow z := z \tilde{\cup} \{e\}$
$y := y \simeq \{e\}$
$e \in x \wedge e \in y \rightarrow z := z \tilde{\cup} \{e\}$
$x := x \simeq \{e\}$
$y := y \simeq \{e\}$
endcase

endloop
end

Program 1.

The problem of updating can be deduced from the above with the correspondence:

x	$\Rightarrow$	t_0
y	$\Rightarrow$	m
z	$\Rightarrow$	t
$x \neq \emptyset \vee y \neq \emptyset$	$\Rightarrow$	$t_0 \neq \emptyset \vee m \neq \emptyset$
$e :\in (x \cup y)$	$\Rightarrow$	$k :\in (K(t_0) \cup K(m))$
$e \in x$	$\Rightarrow$	$k \in K(t_0)$
$e \in y$	$\Rightarrow$	$k \in K(m)$

The corresponding program:

program:
$t := \emptyset$
while $t_0 \neq \emptyset \vee m \neq \emptyset$ **loop**
$k :\in (K(t_0) \cup K(m))$
case when
$k \in K(t_0) \wedge k \notin K(m) \rightarrow S1$
$k \notin K(t_0) \wedge k \in K(m) \rightarrow S2$
$k \in K(t_0) \wedge k \in K(m) \rightarrow S3$
endcase
endloop
end

Program 2.

Let us consider the programs corresponding to the individual choices:

1. If the k key-value occurred only in t_0 - in the old master file - and in the file of transformations did not, this means that m contains no modification for this element. Then $k(t_0)$ should be added unchanged to the new master file and omitted from the old one:

$S1$:
$t := t \cup \{k(t_0)\}$
$t_0 := t_0 - \{k(t_0)\}$

end

Program 3.

2. If the k key-value occurred only in modifier file and in the old master file did not, then only the "insert" transformation can be performed correctly, and the transformation with key k is to be omitted from m:
$S2$:

case when
$k(m).W1 \rightarrow --$ *error*
$k(m).W2 \rightarrow t := t \cup \{(k, k(m).D)\}$
$k(m).W3 \rightarrow --$ *error*
endcase
$m := m - \{k(m)\}$
end

Program 4.

3. If the k key-value occurred both in the old master file and in the file of transformations, then the "deletion" and the "modification" transformations can be performed and the element with key k has to be omitted from both files:
$S3$:

case when
$k(m).W1 \rightarrow SKIP$
$k(m).W2 \rightarrow --$ *error*
$t := t \cup \{k(t_0)\}$
$k(m).W3 \rightarrow t := t \cup \{(k, k(m).G(k(t_0).D)\}$
endcase
$t_0 := t_0 - \{k(t_0)\}$
$m := m - \{k(m)\}$
end

Program 5.

Should the program also issue error signals, these are to be realized in the indicated branches. Let us come back to the original problem where the master file and the modifier file are treated as sequential files and not as sets. Let us assume that the elements of both files are ordered with respect to key-values

and the end of the file is marked by a key-element, called extremal, which is greater than any other key-value.

The program in this case will be:

```
program:
    t := ∅
    dt0, t0 : lopop
    dm, m : lopop
    while dt0.K ≠ extr. ∨ dm.K ≠ extr. loop
      case when
        dt0.K < dm.K → t : hiext(dt0)
                       dt0, t0 : lopop
        dt0.K > dm.K → case when
                         dm.W1 → − − error
                         dm.W2 → t : hiext(dm.K, dm.D)
                         dm.W3 → − − error
                       endcase
                       dm, m : lopop
        dt0.K = dm.K → case when
                       dm.W1 → SKIP
                       dm.W2 → − − error
                                t : hiext(dt0)
                       dm.W3 → t : hiext(dt0.K, dm.D)
                       endcase
                       dt0, t0 : lopop
                       dm, m : lopop
      endcase
    endloop
    t : hiext(extr.)
end
```

Program 6.

Both this solution and the method used essentially agree with those described by Dijkstra.

III.2. Deduction to the problem of elementwise processing

Generalizing the set-theoretical operations - e.g. the union above - and increasing the level of abstraction, one can arrive at the concept of elementwise processing [2]. In this way we can determine the common feature of these tasks which explains why the programs computing the substitution value of functions processable elementwise are always the same. The above task can be deduced to the problem of elementwise processing in two ways:

III.2.A. Deduction to the problem of single-valued elementwise processing

The state space of the problem of updating can also be conceived by forming the union of the sets of keys in the master file and in the modifier file. Then there will be key-values with only one data part belonging to them in the master file, there will be key-values that do not occur in the master file and there will be key-values that occur in both files. From the master file and the modifier file we can thus conceive a file whose elements consist of three components:

a key-value	- from t_0 or from m,
a data part	- from t_0 (which may be "empty") and
a transformation part	- from m (which may be "empty", meaning the identical transformation).

Of course, the two latter fields cannot be "empty" at once. That is:

$X = seq(DX)$, where

$DX = (K, D', V')$

$D' = D \cup \{"empty"\}$

$V' = V \cup \{"empty"\}$

Let $x \in X$. Then $K(x) = K(t_0) \cup K(m)$.

Let us denote by dx an arbitrary element of x. Then:

$$dx.D' = \begin{cases} "empty", & \text{if } dx.K \notin K(t_0) \\ dx.K(t_0).D, & \text{if } dx.K \in (t_0) \end{cases}$$

$$dx.V' = \begin{cases} "empty", & \text{if } dx.K \notin K(m) \\ dx.K(m).V, & \text{if } dx.K \in K(m) \end{cases}$$

Thus applying a state space transformation, we may conceive the task on this state space as a single-valued elementwise processing with the following f function:

$$f(\{e\}) = (e.K, e.V'(e.D'))$$

That is, the transformation part is applied to the data part:

$e.V'."empty"(e.D') = e.D'$
$e.V.W1(e.D') = "empty"$ if $e.D' \neq "empty"$
$e.V.W2("empty") = e.V.W2.D$
$e.V.W3(e.D') = e.V.W3.D$ if $e.D' \neq "empty"$

The specification on this state space:

The state space:

A: $X \times T$
$x \quad t$
B: X
x'

The precondition:

Q : ($x = x'$ and update is simple)

The postcondition:

$R : (t = f(x'))$

The solution:

program:
$t := \emptyset$
while $x \neq \emptyset$ **loop**
$e :\in (x)$
$LET(d, f(\{e\}))$
$t := t \,\tilde{\cup}\, \{d\}$
$x := x \simeq \{e\}$
endloop
end

Program 7.

$LET(d, f(\{e\}))$:
case when
$e.D' \neq "empty" \wedge e.V' = "empty" \rightarrow d := (e.K, e.D')$
$e.D' = "empty" \wedge e.V' \neq "empty" \rightarrow$ **case when**
$e.W1 \rightarrow -- error$
$d := 0$

$e.W2 \rightarrow d := (e.K, e.V'.D)$
$e.W3 \rightarrow -- error$
$d := 0$

endcase

endcase
end

Program 8.

Let us come back to the original state space:

$x \neq \emptyset \quad \Rightarrow t_0 \neq \emptyset \vee m \neq \emptyset$
$e :\in (x) \quad \Rightarrow k :\in (K(t_0) \cup K(m))$
$e.D' \neq "empty" \wedge e.V' = "empty" \quad \Rightarrow k \in K(t_0) \wedge k \notin K(m)$
$e.D' = "empty" \wedge e.V' \notin "empty" \quad \Rightarrow k \notin K(t_0) \wedge k \in K(m)$
$e.D' \notin "empty" \wedge e.V' \notin "empty" \quad \Rightarrow k \in K(t_0) \wedge k \in K(m)$

$t := \tilde{\cup}\{d\} \quad \Rightarrow t := \begin{cases} t, & \text{if } d = \emptyset \\ t \cup \{d\}, & \text{if } d \neq \emptyset \end{cases}$

$x := x \simeq \{e\} \quad \Rightarrow$ if $k \in K(t_0) \wedge k \notin K(m)$:
$t_0 := t_0 - \{k(t_0)\}$
if $k \notin K(t_0) \wedge k \in (m)$:
$m := m - \{k(m)\}$
if $k \in K(t_0) \wedge k \in K(m)$:
$t_0 := t_0 - \{k(t_0)\}$;
$m := m - \{k(m)\}$

The programs $t := t\,\tilde{\cup}\{d\}$ and $x := x \simeq \{e\}$ are independent of each other, so they are interconvertible. Making use of the fact that the branches of conditions in the programs realizing the $d := f(\{e\})$ $(LET(d, f(\{e\})))$ and $x := x \simeq \{e\}$ instructions are identical, and building the union into the branches not containing $d := 0$, we obtain exactly the same program as above.

III.2.B Deduction to the problem of bivariable elementwise processing

It is probably the simplest way of solution to deduce our problem to the problem of bivariable single-valued elementwise processing [2] (or to the problem of bivariable bivalued elementwise processing if we also want to handle the error signals).

The specification:

The state space:

$$\begin{array}{lccccc} A: & T & \times & M & \times & T \\ & t_0 & & m & & t \\ B: & T & \times & M & & \\ & t_0' & & m' & & \end{array}$$

The precondition:

Q : ($t_0 = t_0'$ and $m = m'$ and update is simple and m' is deterministic)

The postcondition:

$R : (t = upd(t_0', m'))$

Because of the special conditions, the updating function is processable elementwise with respect to key-value. It can be given as follows:

$$upd(k, 0) = k(t_0)$$

$$upd(0, k) = \begin{cases} \emptyset, & \text{if } k(m).W1 \\ (k, k(m).D), & \text{if } k(m).W2 \\ \emptyset, & \text{if } k(m).W3 \end{cases}$$

$$upd(k, k) = \begin{cases} \emptyset, & \text{if } k(m).W1 \\ k(t_0), & \text{if } k(m).W2 \\ (k, k(m).D), & \text{if } k(m).W3 \end{cases}$$

The program solving this problem is obtained by a trivial substitution.

IV. The case of multivalued modifier file

If we cannot assume that the modifier file is single-valued, the problem will not be processable elementwise. What can we do? There are two possible ways of abstraction for the solution: (Again, we make use of our files being ordered with respect to key-values.)

IV.1. The data abstraction approach

We perform a state space transformation, and assume that the modifier file has elements of type:

(key-value, sequence of transformations).

That is: $M' = seq(F')$, where $F' = (K, V")$, where $V" = seq(V)$.

With this modifier file we again arrive at an elementwise processing, with the only difference that the appropriate sequence of transformations have to be performed on the original state space:

$upd(k, \emptyset) = k(t_0)$
$upd(\emptyset, k) = (k, k(m).V"("empty"))$
$upd(k, k) = (k, k(m).V"(k(t_0).D))$

(The composition of transformations is meaningful only for an appropriate sequence of transformations because of the domains of definition and the actual ranges of the individual transformations.)

program:
  $t := \emptyset$
  $dt_0, t_0 : lopop$
  $dm, m : lopop$
  while $dt_0.K \neq extr. \vee dm.K \neq extr.$ **loop**
    case when
      $dt_0.K < dm.K \rightarrow t : hiext(dt_0)$
                $dt_0, t_0 : lopop$
      $dt_0.K > dm.K \rightarrow TRANSF("empty")$
      $dt_0.K = dm.K \rightarrow TRANSF(dt_0.D)$
                $dt_0, t_0 : lopop$
    endcase
  endloop
  $t : hiext(extr.)$
end

Program 9.

$TRANSF(par)$:
  $d := par$
  $ak := dm.K$
  while $ak = dm.K$ **loop**
    case when
      $dm.W1 \rightarrow$ **if** $d =$ "$empty$" **then** - - *error*
                else $d :=$ "$empty$"
            endif
      $dm.W2 \rightarrow$ **if** $d =$ "$empty$" **then** $d := dm.D$

```
                                    else - - error
                endif
        dm.W3 → if d = "empty" then - - error
                                    else d := dm.D
                endif
    endcase
    dm.m : lopop
  endloop
  if d ≠ "empty" then t := hiext((ak.d))
  endif
end
```

Program 10.

IV.2. The function abstraction approach

As a rule, the solution of a given problem is inevitably influenced by its specification. If the modifier file is nondeterministic, we can define, starting from the above single-valued elementwise processing, a recursively defined function to evaluate the appropriate sequence of transformations.

The solution with this approach:

The state space:

A :	$X \times T$	the definition of X see above, the only
	$x \quad t$	difference is that the key-values are not unique
B :	X	
	x'	

The preconditon:

Q : ($x = x'$ and x is ordered according the key-values)

The postcondition:

R : $(t = f(x'.dom)_3)$

The definiton of the function f:

$$f : Z \rightarrow K \times D' \times T$$
$$f(0) = (min(x.low.K, extr.), x.low.D', <>)$$

$$f(i+1) = \begin{cases} (f(i)_1, x_{i+1}.V'(f(i)_3), & \text{if } f(i)_1 = x_{i+1}.K \\ (x_{i+1}.K, x_{i+1}.V'(x_{i+1}.D'), & \\ \quad conc(f(i)_3, < (f(i)_1, f(i)_2) >), & \text{if } f(i)_1 \neq x_{i+1}.K \end{cases}$$

The corresponding program will be:

program:
$OPEN(x)$
$LOPOP'(dx, x)$
$t := \emptyset$
while $dx.K \neq extr.$ **loop**
if $f_1 = dx.K$ **then** $ASSIGN(f_2, dx.V'(f_2))$
$f_3 :=<>$
else $k := f_1$
$f_1 := dx.K$
$f_3 :=< (k, f_2) >$
$ASSIGN(f_2, dx.V'(dx.D'))$
endif
$HIEXT'((t, f_3))$
$LOPOP'(dx, x)$
endloop
$HIEXT'(t, f_3)$
end

Program 11.

$LOPOP'(dx, x)$:
if $dm.K = extr. \wedge dt_0.K = extr.$ **then** $dx.K := extr.$
else case when
$dt_0.K < dm.K \rightarrow dx.K := dt_0.K$
$dx.D' := dt_0.D$
$dx.V' :=$ "*empty*"
$dt_0, t_0 : lopop$
$dt_0.K > dm.K \rightarrow dx.K := dm.K$
$dx.K := dm.K$
$dx.D' :=$ "*empty*"
$dx.V' := dm.V$
$dm, m : lopop$

$dt_0.K = dm.K \rightarrow dx.K := dm.K$
$dx.D' := dt_0.D$
$dx.V' := dm.V$
$dt_0, t_0 : lopop$
$dm, m : lopop$

endcase

endif

end

Program 12.

This program essentially differs from the above ones.

$OPEN(x)$:
$dm, m : lopop$
$dt_0, t_0 : lopop$
$f_1 := min(dm.K, dt_0.K)$
if $f_1 = dt_0.K$ **then** $f_2 := dt_0.D$
else $f_2 :=$ "*empty*"
endif
$f_3 :=<>$
end

Program 13.

ASSIGN $(f_2, dx.V'(p))$:
if $dx.V' =$ "*empty*" **then** $f_2 := p$
else case when
$dx.V'.W1 \rightarrow$ **if** $p =$ "*empty*" **then** - - *error*
else $f_2 :=$ "*empty*"
endif
$dx.V'W2 \rightarrow$ **if** $p =$ "*empty*" **then** $f_2 := dx.D$
else - - *error*
endif
$dx.V'W3 \rightarrow$ **if** $p =$ "*empty*" **then** - - *error*
else $f_2 := dx.V(p)$
endif
endcase

endif
end

Program 14.

$HIEXT'(t, f_3)$:
if $f_3 \neq$ "*empty*" **then** t : $hiext(f_3)$
endif
end

Program 15.

V. Summary

Comparing the above methods of solution we can see that a higher level concept usually enables the problem to be treated in a more uniform way. The more efficient the used tool is (in our example the bivariable elementwise processing), the simplier the solution.

In many cases (e.g. with listing problems), the application of data and/or function abstraction offers parallel methods of solution. Of course, the two can also be combined (as illustrated by our latter solution). The choice depends on the programmer, who must be aware of the possibilities and should prefer the solution which can better be generalized.

References

[1] **Dijkstra E.W.,** *A discipline of programming,* Prentice-Hall Inc., Englewood Cliffs (N.Y), 1976.

[2] **Fóthi Á.,** *Bevezetés a programozáshoz,* Tankönyvkiadó, Budapest, 1983.

[3] **Fóthi Á.,** A mathematical approach to programming, *Annales Univ. Sci. Bud. Sect. Comp.,* **9** (1988), 105-114.

[4] **Naur P.,** Programming as theory building, *Microprocessing and Microprogramming,* **15** (5) (1985), 253-261.

[5] **Floyd C.,** Eine Untersuchung von Software-Entwicklungsmethoden, *Programmierumgebungen und Compiler, Tagung I/1984 des German Chapter of the ACM,* eds. H.Morgenbrod und W.Sammer, Teubner Verlag, Stuttgart, 1984, 248-274.

[6] **Kozma L. and Varga L.,** A methodology for the development of shared object classes, *Proc. of Int. Conf. on Applied Informatics, Eger, Hungary, 23-26 Aug. 1993,* 5-14.

(Received December 14, 1993)

Á. Fóthi, J. Nyéky-Gaizler and É. Harangozó
Department of General Computer Science
Eötvös Loránd University
Pázmány Péter sét. 1/D.
H-1117 Budapest, Hungary