CONTEXT-ORIENTED PROGRAMMING

M.L.Gassanenko
SPIIRAN, St.Petersburg, Russia
mlg@forth.org, mlg@iias.spb.su

ABSTRACT
Context-oriented programming (COP) introduces one more notion to reason about the structure of software systems: a context (an environment) is a set of entities bound with a system of relations. This view is applicable where the object-oriented one is inadequate. Implementation of COP requires the same techniques as OOP: COP and OOP are different things assembled from the same components. COP allows things that OOP cannot do, for example, COP enables us to use late binding for elementary data that are not OOP objects.

Keywords: Data-Driven Approach, Object-Oriented Programming, Context-Oriented Programming, Virtual Method Tables, Forth, data structures, contexts.

Abbreviations

DDA	- Data-Driven Approach, or data execution
OO	- Object-Oriented
OOP	- Object-Oriented Programming
COP	- Context-Oriented Programming
VMT	- Virtual Method Table

1. The purpose of the work

The key element of software activities (like designing, writing, debugging, reading and modifying programs) is reasoning. We do not give a definition for reasoning, but we assume that the use of a small number of adequate concepts facilitates reasoning—that is, if the concepts in use are adequate and expressive, if no auxiliary concepts are required, it is easier to reason about programs.

The purpose of this work is to extend the set of notions, which may be used to reason about the problem to be solved and the structure of the corresponding software system.

Along with the notion of a context, an approach to problem decomposition, and a programming technique appear.

2. The method of COP

2.1. Quick introduction

An environment (a context) expresses the idea of a system of relations (where "relation" is used in its everyday meaning, a connexion between things). The phrase "system of relations" implies that there are some entities, upon which relations are established. The key point in COP is that to *use* the relations, we *do not need to know details* about the entities. This is good, but how can we use the relations? In principle, different approaches are possible. The simplest way is to implement a set of functions, which imply that the data elements to be processed are bound with these relations. (Indeed, using these functions, we use knowledge of that the data elements are related, otherwise it would be incorect to use the functions). The next step is to introduce polymorphism. This may be done via tables of pointers to functions, that is, virtual method tables (VMT, where "method" is understood as "member function"). *Contexts are implemented as VMT's, and these VMT's are not bound to data.* So, from the implementation point of view, there are interchangeable sets of functions, allowing us to work with different (but somehow similar) sets of objects; the objects in a set do not necessarily form a datastructure, and may have different lifetimes.

It should be also noted that COP introduces *two levels* at which we can work with contexts. *At the level of implementation,* we know all details about related entities that constitute the context. *At the level of use,* we do not have full access to the entities; we can use only a set of functions allowing us to work with the entities, but this should be enough to accomplish our tasks. In exchange, this restricted access allows us to work in different contexts (of the same "class") *in a uniform way.*

2.2. Some examples

Example 1. A Forth compiler is able to generate both threaded and native code. It includes two implementations of the set of compiling words (COMPILE, , >MARK , etc.): one for the threaded code mode and another for the native code mode. The polymorphism of the compiling words is implemented via VMT's: compiling words are "messages", each one calls the corresponding method from the current VMT (the VMT of the current context). Knowledge about the structure of code being generated is concentrated in these methods.

May these two modes of code generation be considered as OOP objects? Yes, but they would be very strange objects.

1) They do not have their own data; instead, they use the same Forth dictionary space. The dictionary space is not associated with a VMT, and VMT's (classes?) change while the program executes.

2) The words >MARK (-- *orig*) and >RESOLVE (*orig* --) construct a reference to the branch destination for a control transfer instruction. In the two compilation modes, both references and the *orig* values that identify them have different data types: the references have different formats, and the *orig* values, playing the same role, have different sizes on stack.

3) The data elements for which late binding is used come and go (e.g. the *orig* values used by >RESOLVE). Their actual types are determined at run-time, but not because these values are objects knowing their type. The type is known from the context.

4) The data processed within a context do not form a datastructure (e.g. *orig*'s are kept on the stack).

5) Data which type is known only within a context are visible and manipulated from outside the context.

6) If the native code mode and the threaded code mode are analogous to OOP classes, only one instance of each such class is meaningful, which is a bit unnatural for OOP.

It is not what we usually recognize as an object.

Example 2. We have to generate target code optimized on memory. The target instruction set includes a number of short, medium and long control transfer instructions (opcodes do not necessarily occupy a whole byte). To generate short instructions where possible, we decide to use intermediate code and the following algorithm: first assume all forward references to be short, and try to generate code; if some references cannot be resolved, assume them to be longer (medium, then long), and try to generate code again; repeat this until all references are resolved. Since Forth definitions are usually small, this algorithm is acceptable.

In this example a control transfer instruction (in the intermediate code) is an object which changes its class (short/medium/long) in order to make interactions with other objects correct. Or, from a different point of view, we assemble an OOP-like object from a context and a data record at run-time.

1) An object can change its class.

2) Run-time class recognition is possible.

3) The class reflects the role of the object, rather than its internals. The class of the object is changed not because of the internal causes, but because of other objects.

4) The same object belongs to multiple classes: according to its purpose (a conditional/unconditional branch, a call, an OF run-time semantics) and according to its size (short/medium/long).

All this is possible because a context (that is, its VMT) is not bound to the data.

Is it an OOP example as well? Yes. Inside an OOP object, there is a context (a set of interrelated components). In this example we bound data records and contexts by means of the programming language, that is, we used contexts to implement a variation of OOP. Contexts are a powerful expressive means in a programming language, they may be used to implement OOP and DDA.

Let us summarize the mentioned differences between COP and OOP:

- an OOP object is a unit; a context is not a single unit
- contexts do not have instance data
- the data that are processed within a context do not form a datastructure
- a context is not bound to the data that are processed within it
- COP enables us to manipulate data elements (that are not OOP objects and have no type information) not knowing their type
- COP enables us to use late binding on such data elements

- the lifetime of data elements does not depend on establishing and de-establishing contexts, and vice versa: data may persist while contexts come and go, and contexts may persist while data come and go
- multiple contexts may be current at the same time
- manipulations with contexts are an important *constituent part* of OOP
- for an object, run-time class recognition is possible (it is possible to switch to a different context)
- the class reflects the role of the object, rather than its internals
- the same object may belong to multiple classes (probably, dynamically determined)

2.3. Definitions

We define a *context* as a set of entities bound with a system of relations. We propose to work with a context at two levels. At *the level of use* (from "the outside view"), we know nothing about the entities except that they are related in some specific way, and we can manipulate the entities using that knowledge. At *the level of implementation* (from "the inside view"), we know all details about the entities, but we use this level only to implement the desired system of relations.

Let us compare this to OOP. An object is a single unit having some properties (typically implemented via member functions). At the level of use, we know nothing about the internals of the object, but we can manipulate it knowing its properties. At the level of implementation, we have a set of components bound by a system of relations (that is, a context), this context implements the properties of the object. At the level of use, this context is "encased into a shell", which makes it a single unit.

So, there are three levels: a unit—a group of unrelated units—a context—a unit. COP enables us to work at the levels of a group of unrelated units and of a context. OOP enables us to work at the levels of a context (a set of related units) and of a single unit (that is, an object).

A *method* of a context is a function defined in the context; different contexts may provide different definitions for the same method. A *class* of contexts is a set of interchangeable contexts implementing the same set of methods (a defining word for contexts of the same class also may be called so). There is a notion of *the current context*. In each class, there is only one current context. In the simplest case, this assumes that for each class there is a pointer to the current context. In the current implementation such pointers are global, but, generally speaking, they need not be global, they must have an adequate visibility. A *message* is a function that calls a method that corresponds to it *from the current context* of the class associated with the message. Messages defined for different classes must have different names. *Inheritance* is defined for contexts *of the same class*. In the current VMT-based implementation, the word `INHERIT` is just a VMT copy routine (when inheritance is performed, some methods are usually redefined).

If the pointer to the current context of some class is global, routines that save and restore the current context may be required, to support nesting of operations that employ contexts of this class. If such pointers may be local, it is possible to implement nesting of contexts of the same class via local pointers. Another approach to context nesting is to have a stack of pointers to the contexts. A message examines the context stack, finds the topmost context of the required class, and calls the corresponding method from this context.

2.4. COP and existing techniques

Contexts are similar to both classes and objects, probably, the most close match are objects in prototype-based OO languages [UnS87]. Like classes, contexts have VMT's and have no instance data. Unlike classes, they have no instances. Unlike classes, they (not their instances) are manipulated in a program. An OOP object is put in a shell that separates the object from the outside world; an arbitrary piece of code cannot penetrate the shell and execute within the object's context. Contexts are open to such invasion.

Contexts (that is, sets of related entities) may be found in existing programming languages, but typically they are not interchangeable with other contexts of the same type.

- A library is a context (a set of related functions).
- Function call arguments form a context when the function is called. From the application point of view, the arguments are related because they participate in the process of result calculation (we know *the meaning* of each parameter, different parameters have different meanings, but they are related). From the language implementation standpoint, the arguments just have the same lifetime and visibility.

- Inside an OOP object, there is a context (a set of related variables and functions).
- A system of OOP objects may form a context. For example, the Microsoft C++ compiler generates about 50k of source code for a new MFC application. This code is a set of classes bound with relations—a framework upon which a new Windows application is buit. This framework is a context; the initially established relations do not change in program development. Can this framework benefit from COP? Probably, no, at least at the level of a whole Windows application. At first, the problem is already solved. At second, COP assumes that we work with contexts at two levels: the level of implementation and the level of use. In case of a MFC application, programmer works only at the implementation level, and cannot benefit from ability to use different contexts (applications) in a uniform way, because the code that copes with the whole application is already written. Such construction of a context has at least two disadvantages: at first, the system relations are attributed to objects; at second, there is no classification of relations between the objects, whereas we may wish to consider the system at different levels, with different sets of relations relevant at each level.
- A Forth wordlist typically is a context (the F-PC wordlist `HIDDEN` containing auxiliary functions from all subsystems is an exception).
- A record may implement a context (not surprizing, if a procedure activation record can do this).
- Lisp "association lists" of (<*symbol*>.<*value*>) pairs can implement contexts.

Why COP is similar to OOP? Both problem formulation, the approaches to solution, and implementation techniques were, for the most part, borrowed from OOP. We repeated the steps that lead from the concept of an object to OOP, but we started from a different concept (a context rather than an object), and the result is different from OOP. Is it necessary to copy OOP solutions in COP? Probably, no. Practice showed that implementation techniques that are perfect in case of OOP, may be not so good for COP. Another arbitrary choice that we made is the use of sets of functions to implement (that is, provide a way to use) the relations between the context components; other approaches may be applicable here.

Let us give an example of how a technique which is good in case of OOP (namely, VMT's) may be not so good for COP. Let us return to the example of Forth compiler able to generate both native and threaded code. Most control structure words are sensitive to the compiler mode (native/threaded code). We have to implement polymorphism ourselves, but the implementation must be simple, because COP is not the ultimate goal but an auxiliary means. It is undesirable to implement compiling words via VMT's, since they are too many, are not executed often, and cannot be predeclared (the user is allowed to add his own compiling words), while the requirement for VMT extensibility unduly complicates the implementation. It is undesirable to place definitions of compiling words into the section of context methods, because the words like `>MARK` and the words like `IF` definitely belong to different levels. The words like `>MARK` reflect peculiarities of the target code, the compiling words like `IF` specify how control structures are mapped onto the target code. So, we assemble the words like `>MARK` into a context (implemented via VMT's), and consider files with definitions of compiling words as packages aware about context details. To implement polymorphism of compiling words, we decide to use a variation of a CASE statement (syntactically, it looks as a set of definitions with the same name, each beginning with a special word, but executes like nested IF statements) and a variable indicating the current context. The loss of execution speed is not significant, while implementation simplicity is an advantage. So, in this instance the CASE-based approach offers an advantage over the VMT-based one.

Let us take one more look at this very interesting example. We implement *two* contexts that form a *hierarchy*. The first context (with the words like `>MARK`) is at the lower level, providing services for the second context (with the compiling words) being at the higher level. These contexts implement *different* systems of relations: in the first case, the words like `COMPILE,` , `>MARK` and `>RESOLVE` allow generation of target code in terms of tokens and references; in the second case, compiling words (like `IF` and `THEN`) allow us to use control structures. Our expressive means have been sufficient to underline the difference between these two levels.

COP may be implemented on an OO language. It is possible (and did happen in practice) that the code will not have much meaning from the OOP point of view and a lot of comments will be required to explain what is really meant.

Let us give a few examples. The first, artificial, example illustrates how the context corresponding to the data may be established as a result of analysis of data in an OO language: a procedure receives a set of data, analyses the data, determines how they are related, and creates an object of one of several possible classes, the object constructor parameters are those of the procedure.

Then, the data are processed in the context of that object. When this is done, the object is destroyed. Aside from the large number of auxiliary entities, one of the most annoying problems is how to name the classes.

The next example is taken from real life. Contexts are implemented as C++ objects without data. There is a pointer to the current context. A number of global inline functions is defined to call the current context methods using that pointer.

2.5. Miscellaneous notes

A context is a set of related units. Let us note three particular kinds of reasons why the units may be related (the list does not exhaust the possibilities):

a) *a set of statements* specifies relations between the units (a mathematical example: a group is a set with an operation, the operation and the set elements are related in the way specified by the group axioms; note that the operation is one of the related entities)

b) the units are involved in *the same process* (e.g. two files contain configuration information used by an application)

c) the set of units is a part of *a hierarchy*, that is, the set of units uses lower-level services and provides services for the higher level (so is the BIOS set of i/o functions, the lower level being hardware and the higher level being OS).

An interesting kind of context modifiers was proposed in [Gas93]. "Inflectors" are a special kind of contexts: when an inflector is placed on the context stack, it modifies the last context of the corresponding class by specifying its own reaction on some (but not all) messages. That is, the inflector intercepts some messages while others are passed to the "inflected" context. We have to note that up to now inflectors have not been applied to practicall problems, although in one case they could be used (four contexts could be replaced by two contexts and one inflector). Inflectors have not been used mainly because they require a stack of contexts, whereas the existing implementation used pointers to the current contexts. If the inflector allowed to replace 12 contexts by 6 contexts and one inflector, the results would probably justify the efforts.

2.6. COP and DDA

The technique of data execution (Data-Driven Approach) was independently invented at both sides of the 'Iron Curtain'. The methods of data execution trace their origin to macroprocessors of 1960-es: it was found that it is possible to define data as macros, and execute these data with a macroprocesor.

Data execution may be applied where data elements must be processed according to their type. Data may have a hierarchical structure. The number of data types must be finite. For data execution,

```
data type  =  how data must be processed
```

In all variations of DDA, a data interpreter is created. It is able to repeatedly process data items according to their type. A context is created which specifies how data of different types must be processed. Then, data are put into that context and interpreted with the data interpreter. The objective of data processing is achieved via such execution of data.

The Western version is usually Lisp-based [HyS90], although there is a report on a successful application of the Lisp methods to Forth [Smi91]. On the other hand, there are Forth techniques [Rod90][Koo90] that do not have roots in Lisp. The Forth language has always been using data execution: generation of threaded code (with the help of immediate words), and Forth Assemblers are examples of data execution. In both these examples a definition is a program that generates the desired code. Along with the threaded code, this is macroprocessors' heritage.

The Russian version of data execution [Tuz88][Tuz90] uses data compilation. Data are transformed into a universal executable form, which is a particular case of threaded code. The data interpreter is the inner interpreter of Forth.

Here is a simplified example in the [Tuz88] spirit.

```
\ data types
DEFER LS     \ list
DEFER NUM    \ number
DEFER STR    \ string
\ Two lists
: list1   2 NUM   3 NUM   C" abc" STR ;
: list2  10 NUM   ['] list1 LS   40 NUM   C" end" STR ;
\ Print a list
: .str COUNT TYPE ;
: .ls  ." ( " EXECUTE ." ) " ;
: print ( addr -- )
      ( save-current-context)
      ['] .   IS NUM
      ['] .str IS STR
      ['] .ls  IS LS
```

```
                ." ( " EXECUTE ." ) "
        ( restore-current-context)
;
\ Count numbers in a list
: DROP_1+ DROP 1+ ;
: count-numbers-deep ( addr -- n )
        ( save-current-context)
        ['] DROP     IS STR
        ['] DROP_1+  IS NUM
        ['] EXECUTE  IS LS
               0 SWAP EXECUTE
        ( restore-current-context)
;
```

A sample session output:

```
' list1 print
( 2 3 abc)
' list2 print
( 10 ( 2 3 abc) 40 end)
' list1 count-numbers-deep .
2
' list2 count-numbers-deep .
4
```

In the Tuzov's work, the functions that represent lists are unnamed and reside in a special garbage-collected area; the context information (the values assigned to the DEFER words) is saved at the return stack. You see that operations with contexts are an important part of the data execution method.

Data execution is based on interaction between a *process* and a *context*. The process is responsible for looking over the data elements; the context is responsible for actions that should be performed on data elements of each type. The process, in turn, is a result of interaction between the *data interpreter* and the *executable data* (for the data, representation and meaning may be considered).

3. Implementation

There are several ways to implement polymorphism of context methods. Three approaches are considered:

1. Virtual method tables
2. A sort of CASE statement
3. Functional variables (DEFER's)

All these have been used in practice. One more possible approach is the use of Forth wordlists.

3.1. Implementation via functional variables (DEFER's)

Polymorphism may be implemented using variables that return values assigned to them (QUAN or VALUE) and perform functions assigned to them (VECT or DEFER). When a context is set, these variables receive values corresponding to the context. This approach was used in [Tuz88], [Tuz90].

The word DEFER (in some systems, VECT) creates a functional variable: a word that executes the word whose execution token *(xt)* is stored to its data field. A special prefix TO (in some systems, IS) is used to assign a new *xt* to such variable. The data variables are defined using the word VALUE (in some systems, QUAN), the assignment syntax is the same as for DEFER's.

An example:

```
\ Context parameters
0 VALUE first-unit  ( -- n )
0 VALUE last-unit   ( -- n )
0 VALUE unitN       ( -- addr )
  DEFER unit+       ( addr -- addr' )
  DEFER units       ( n -- n' )
  DEFER unit<       ( n1 n2 -- f )
\ Contexts
VARIABLE A-units-N
: contextA
        0           TO first-unit
        100h        TO last-unit
        A-units-N   TO unitN
        ['] 1+      TO unit+
        ['] NOOP    TO units
        ['] <       TO unit<
;
```

```
VARIABLE B-units-N
: contextB
      200h        TO first-unit
      0           TO last-unit
      B-units-N   TO unitN
      ['] 2-      TO unit+
      ['] 2*      TO units
      ['] >       TO unit<
;
\ Code that works in both contexts
: new-unit-id ( -- n )
      unitN @ unit+ DUP unitN !
      unitN @ last-unit unit< 0= ABORT" units space exhausted"
;
```

Inheritance is also possible:

```
: contextA1
      contextA                \ like contextA, but
      80h         IS last-unit \ last-unit is different
;
```

The advantages of this approach are that

1) it requires no special means
2) DEFER's are a bit faster than calls via a table

The disadvantage of this approach is a large number of auxiliary names: each word to be assigned to a DEFER must have a name. If the number of auxiliary definitions is large, this approach leads to cumbersome (read "less readable and maintainable") code.

3.2. Implementation via a sort of CASE

Suppose that we have an algorithm, which must work in three modes: mode-A, mode-B and mode-C. It is obviously possible to implement functions which behavior depends on the current mode via a CASE statement and a variable, which holds the number of the current mode. (We use the term "mode of operation" to underline that the function itself contains all information about its different behavoiurs; the result is the same as in the case of a context containing its methods.) The straightforward approach,

```
: foo
      CASE CurrentMode
      Mode-A OF ... ENDOF
      Mode-B OF ... ENDOF
      Mode-C OF ... ENDOF
      1 ABORT" polymorphous function not defined for this mode"
      END-CASE ;
```

has an obvious pitfall: the code is unreadable and unmaintainable. We prefer to have 3 small separate definitions instead of this bloated CASE. So, we choose the following syntax:

```
: foo
    [modeA-only]
    ... ;
: foo
    [modeB-only]
    ... ;
: foo
    [modeC-only]
    ... ;
```

Note that these mode-dependent definitions may be located (and usually are located) in different files.

The mode specifier words (like [modeA-only]) compile a sort of CASE statement across these definitions. If the current mode variable contains a value different from that for the modes A-C, an error message is generated. A mode specifier may check against several modes, e.g.:

```
: foo
      [modesB&C-only]
      ... ;
```

The word [default] enables one to specify actions to be performed when the current mode is not recognized by other definitions of the same name. The order in which the definitions with mode specifiers appear is not important. A definition without a mode specifier never becomes the default action; instead, a warning "was unpolymorphous" is issued. A polymorphous definition may be freely used after its first

mode-dependent definition; if more mode-dependent definitions are added, the functionality of the definition extends even in the code to which this definition is compiled.

That is, we may have definitions,

```
: IF [forth-only] COMPILE ?BRANCH >MARK ; IMMEDIATE
: WHILE [COMPILE] IF CS-SWAP ; IMMEDIATE
```

and if we extend the functionality of `IF` to, say,

```
: IF [asm-only] ( comp: opc -- ) C, rel8>MARK ; IMMEDIATE
```

then the functionality of `WHILE` is also extended (it can work in the `[asm-only]` mode).

The described method facilitates reuse of existing code, especially compared to the bloated CASE approach: although the ideal way of reuse is no changes, addition of a single line that cannot change the stack effect is still acceptable. When a need to reuse the code will appear again, it will be possible to define unnecessary mode specifiers as immediate no-ops.

The requirement that the code is reused without rewriting poses one more restriction on the implementation: the contents of the return stack in a definition with a mode specifier must be the same as for an ordinary colon definition. We require this because existing code sometimes uses return addresses manipulations, typically, `RDROP EXIT` to exit two procedures. This is non-standard, but this worked for years, and there's neither need nor time to redesign it.

There are the following ambiguous conditions:

- a mode specifier is not the first word in a definition;
- one mode-dependent definition of a name is immediate while another is not;
- two or more `[default]` versions of the same name.

It would be a good style to report errors for all these cases.

It should be also noted that the syntax itself does not assume any particular way of implementation: it may implement the DEFER and the VMT approaches as well, provided that access to the dictionary structures is possible.

We have to note that there is no ANS Forth implementation of such mode specifiers. To implement them, we need access to the dictionary headers and to the generated code; in addition, we need a word allowing us to exit the current colon definition and execute another colon definition.

Although it is possible to implement extensible definitions by means of ANS Forth, the syntax, e.g.

```
:NONAME
        ...
; mode-A version-of foo
```

is much less readable and requires more serious changes in the source code. The safety of code reuse and readability are more important than portability of the compiler extensions, therefore we prefer the approach of the compile-time mode specifiers.

The current implementation. In the current implementation, mode-specific versions of a polymorphous word form a chain. When a mode-dependent word is executed, execution proceeds along this chain until a version that matches the current mode is found. The last procedure in the chain is either a `[default]` definition or a error report, in both cases it matches any mode.

A mode specifier compiles guarding code, which will check the mode variable and, if the current mode is different from that corresponding to the specifier, exit the current colon definition and execute the next definition in the chain (specified as a parameter of the guarding word). The latter definition is either another mode-dependent definition of the same name or a word that reports an error. If the current mode matches the specifier, the guarding code has the effect of NOOP.

A mode specifier is an immediate word that compiles a guard word along with its parameter. In the current version guard words fetch their parameters from threaded code, in the same way as LIT does. A mode specifier performs the following actions:

compile a guard word
obtain the name of the current colon definition
find the name (if the name is found, its xt is obtained)
if the name is found
  if it is a mode-dependent definition (its first word is a guard),
    remove the header of the current definition and switch to the `:NONAME` mode
    modify the code of the found definition so that the current definition
      becomes the second definition in the chain (the first one is the found definition)
  else
    print the warning that the name was unpolymorphous
    compile xt of the error-reporting word as a parameter
  endif
else
    compile xt of the error-reporting word as a parameter
endif

The word `[default]` has some specifics. It compiles a guard which always passes control to the procedure specified by the guard's parameter, finishes the current definition and begins a `:NONAME` definition; this definition becomes the last procedure in the chain, replacing the error-reporting routine. This procedure can never be the first one in the chain, which is important because the mode specifiers cannot change the first element in the chain.

3.3. Implementation via virtual method tables

There is a defining word for VMT classes. A VMT class is also a defining word, it creates a VMT word, which implements a context (a virtual method table is a table which contains function addresses). Each VMT class has a pointer to the current VMT of this class. When a VMT word is executed, its VMT becomes the current VMT of this class.

A VMT class has a counter of messages. Immediately after creation of a VMT class, its messages must be declared. In the current implementation, it is not possible to extend the set of messages after creation of one more VMT class. The names of member functions must be unique. Since there may be multiple context classes, multiple contexts may be current (each in its class), and there is no way to decide what was meant if some message is defined for two classes.

An example. This code was used to test the VMT implementation. The virtual method tables are called here "operation tables" (OPTAB's)

```
OPTAB-CLASS class1  \ declare a VMT class
    OPER xx         \ and its member functions
    OPER yy
    OPER zz
class1 t1           \ t1 is a VMT of the class class1
OP: xx ." xx " ;    \ definitions of member functions
OP: yy ." yy " ;
OP: zz ." zz " ;
class1 t2           \ t2 is another VMT of the class class1
OP: xx ." XX " ;    \ definitions of member functions
OP: yy ." YY " ;    \ (unlike those of t1, they print in uppercase)
OP: zz ." ZZ " ;
```

There are VMT classes with initialization: when the VMT becomes the current one, it executes its INIT operation. A special defining word is necessary for this operation because it may be defined in multiple classes. (To be more precise, INIT is not a name of a member function. Each such class has an unnamed function which it executes when becomes current. The word `:INIT` is used to define such functions.)

```
OPTAB-CLASS+INIT class2     \ contexts of class2 have an INIT operation
    OPER xxx
    OPER yyy
    OPER zzz
class2 u1
OP: xxx ." xxx " ;          \ print in lowercase
OP: yyy ." yyy " ;
OP: zzz ." zzz " ;
:INIT ." u1: " ;
class2 u2
OP: xxx ." XXX " ;          \ unlike u1 methods, print in uppercase
OP: yyy ." YYY " ;
OP: zzz ." ZZZ " ;
:INIT ." u2: " ;
```

Inheritance:

```
class2 u2a
u2 INHERIT                  \ u2a is like u2, but
OP: xxx ." *** " ;          \ the xxx member function is different
```

We have to note that the ANS Forth's specification of `:NONAME` (-- *xt colon-sys*) is inconvenient because the size of *colon-sys* is implementation dependent. One has either to define a word like `;OP` (cumbersome and unnecessary) or to write a piece of system-dependent code. In our case this has been the word `START:` (-- *colon-sys xt*) different from `:NONAME` only in its stack effect.

4. Practical use

The first work on COP has been [Gas93], but it was not until 1997 that COP has been used to solve practical problems.

COP has been used in two projects. The first one is in C++. Contexts have been implemented as objects with no data. In one place it has been desirable to let an algorithm function in several modes (creation of an isolated object, creation of an object bound to other objects, and the error mode). When an error is detected (due to e.g. invalid connections to other objects), the algorithm has to switch to the error mode. When the algorithm became a two-screen mess of 'if' and 'switch' statements, the author decided to

use COP, and the code become comprehensible. After that, the author wrote about a screen of comments explaining how this works (this would not be necessary if COP was a common practice).

The second project has been a Forth cross-compiler for i8051. An existing compiler into subroutine-threaded code (STC) had to be modified to allow generation of code of a nontrivial structure that could be executed from RAM. From the reuse considerations, it was decided to keep the ability to generate STC. As a result, the compiler can work in different modes. These modes are implemented via contexts. There have been a number of challenges; in all, DDA was used twice and COP was used 5 times (this is the number of VMT classes, including the two used for DDA). I can conclude that the combined use of COP, DDA and the traditional procedural programming (to be more precise, of its Forth version) *is* convenient.

5. Related work

I would like to mention two works as predecessors of COP. At first, it is the method of data execution [Tuz88] [Tuz90]. In this approach, an universal executable data representation is used (a complex data structure is represented as a function that processes its data elements, that is, knowledge on how to process this, probably unique, data set may be found in the data set itself); to perform an operation on such data, a context is established such that execution of the data set will perform the operation; then data are executed (the executable data set is responsible for looking through the data elements, and the context specifies what to do with them). Context manipulations are an important component part of this method.

At second, it is the prototype-based OO language SELF [UnS87], which objects more resemble contexts (both Forth wordlists and nested local contexts of procedures) than data records:

1) there are no classes, an object itself holds its methods
2) a SELF object has no explicit variables, variables are emulated via two (fetch and store) methods
3) when an object cannot respond to a message, it passes the message to the next object (which resembles visibility rules for variables in nested blocks).

COP has been factored out from these two approaches.

6. Conclusion

Context-Oriented programming is one more way to think about the software systems. A context expresses the idea of a system of relations. Contexts resemble both OOP classes and OOP objects, but are different from them. COP complements OOP, working in cases where OOP is inadequate. Contexts provide a means to implement customized variations of OOP and another important technique—Data-Driven Approach (data execution).

References

[Gas93] Gassanenko, M.L. Context-Oriented Programing: Evolution of Vocabularies. Proc. of the euroFORTH'93 conference, 15-18 October 1993, Marianske Lazne (Marienbad), Czech Republic, 14 p.

[HyS90] E.Hyvõnen, J.Seppãnen. Mir Lispa (2 volumes). Moscow: Mir, 1990. ("The world of Lisp", in Russian)

[Koo90] Koopman, Philip. "TIGRE: Combinator Graph Reduction On The RTX2000", Proc. of the 1990 Rochester Forth Conf., p.82-86.

[Rod90] Rodriguez, B. J. Rules Evaluation through Program Execution. Proc. of the 1990 Rochester Forth Conf., 1990, p.123-125.

[Smi91] Smith N.E., 1991. Data driven Applications in Forth.//Proc. of the 1991 Rochester conf., p.100-101.

[Tuz88] Tuzov V.A. Funktsional'nye metody programmirovanija./ Instrumentalnye sredstva podderzhki programmirovanija - L:LIIAN, 1988. - p.129-143. (The Functional Methods of Programming [data execution is a powerful programming technique], in Russian)

[Tuz90] Tuzov V.A. Jazyki predstavlenija znanij. - Leningrad:LGU, 1990. (The Languages of Knowledge Representation [what an ideal language of knowledge representation should be], in Russian.)

[UnS87] David Ungar, Randall B. Smith, "SELF: the Power of Simplicity", OOPSLA'87 Conference Proceedings, SIGPLAN Notices, v.22, n.12, December 1987.