DEFINING WORDS WITH CLASS

Alan T. Furman

1634 Roll Street
Santa Clara, CA 95050

ABSTRACT

Defining words can be implemented that collect new words in a set or class as they are being created. One can then automatically apply functions to the entire class. This technique leads to more concise and expressive code, and works very synergistically with symbolic self-labeling output. It finds practical use with surprising frequency.

INTRODUCTION

Very often one performs the same operation on a number of similar things: executing a sequence of pass-fail tests, clearing variables, displaying parameters, polling input bits, and so on. Going by conventional programming-language insight, there are only two ways to do this. The first is to have a Forth word for each thing. Then, to operate on all of them, explicitly invoke the first word, the operation, the second word, the operation...etc. The second is to make an array of things. To operate on all of them, write a DO loop. Neither approach is appealing. The first is laborious and verbose, and requires a lot of editing when adding or removing things. The second sacrifices the benefits of having a Forth word for each thing: the ability to invoke, test, read, etc. each thing symbolically.

Since Forth gives the programmer control over the mechanism of word creation, one can use a technique that combines the advantages of both of the above methods. This paper describes tools for creating words that have both individual and group identity. I will use the term "class" for such a group. (It is not the same usage of the term as a "class" in object-handling environments such as the Smalltalk language.) The result is an easy way of automatically converting operations on individual class members to operations on the whole class, without further testing required.

The next section describes the words with which the technique is implemented. It is followed by a simple example of the technique's use. The paper concludes with some other suggested applications, and general observations.

THE TECHNIQUE

The words CLASS , >CLASS , and APPLY suffice for nearly all practical cases. They are defined as follows.

```
: CELLS ( #cells -- address_increment )
    2 * ;

: CLASS ( max#cells -- )
    CREATE 0 , DUP , CELLS ALLOT ;

: INCSIZE ( class_pfa -- )
    1 SWAP +! ;

: ?CLASSFULL ( class_pfa -- )
    DUP @ SWAP 1 CELLS + @ < 0=
    IF ." CLASS OVERFLOW" ABORT THEN ;

: LATESTCELL ( class_pfa -- address )
    DUP 2 + CELLS + ;

: >CLASS ( class_pfa -- )
    DUP ?CLASSFULL DUP INCSIZE LATESTCELL HERE SWAP ! ;

: MEMBER@ ( class_pfa index -- n )
    2 + CELLS + @ ;

: APPLY ( address class_pfa -- )
    SWAP OVER @ 0 DO
        OVER I MEMBER@ OVER EXECUTE
    LOOP 2DROP ;
```

The code above follows the 83 standard. For 32-bit systems, replace the 2 in the definition of CELLS by 4.

The defining word CLASS creates a Forth word that is like a string of cells ("string" meaning a one-dimensional array with a built-in element count). It is initially empty, but it can accomodate up to "max#" members. The first two cells in a CLASS word's parameter field are, respectively, the number of members and the capacity. The remainder are pointers to the members' parameter fields.

Elements are added using the word >CLASS which, when embedded in a defining word, gives the latter's offspring membership in a class. In the case of a class called WHATEVER we use >CLASS thus:

```
: DEFININGWORD
    CREATE  WHATEVER >CLASS ...
    ...
```

>CLASS reads HERE, which points to the pfa of the word just made by CREATE .

APPLY takes a word's execution address and a class as arguments. The word is applied to each member of the class. More specifically, it is executed with a pointer to each member's pfa on the stack. For example, suppose the word MUMBLE operates on the pfa of a single member of the class WHATEVER . Then

```
' MUMBLE WHATEVER APPLY
```

will invoke MUMBLE on each member of the WHATEVER class.

AN EXAMPLE

The user of a simulator wants to learn the system's behavior for various inputs and system parameters. If the system is an automatic motorized positioner, typical parameters might be load inertia, feedback controller gain, and output voltage swing of the motor driver. The user will vary these numbers while re-running the simulator, and will often want to have all their values displayed. This example will demonstrate how convenient classes can be for displaying these parameters.

The first step is to create the class PARAMS .

```
10 CLASS PARAMS
```

allows room for up to 10 members. Next, use the compiler word PARAM to create the needed words:

```
: PARAM
    CREATE  PARAMS >CLASS
    1 CELLS ALLOT ;

PARAM INERTIA
PARAM GAIN
PARAM VOLTAGE
```

By itself, a PARAM behaves in the same manner as a variable. The user can interactively read or alter it by name.

We now turn to the matter of displaying the value of these parameters. First we define a word .PARAM

```
: .PARAM ( pfa -- )
    CR DUP .NAME ."  = " ? ;
```

This word generates a labeled output of a variable's value. The definition of .NAME is system-dependant; NFA COUNT TYPE is a typical implementation.

The word .PARAM may be used--and tested--with an individual PARAM in, for example, this way: typing

```
10 GAIN !    GAIN .PARAM
```

will give the output GAIN = 10 . To display all of the params requires only the following:

```
: .PARAMS ['] .PARAM PARAMS APPLY ;
```

Here redundancy has been eliminated in two stages: first, with self labeling output (replacing ." GAIN = " GAIN ? by GAIN .PARAM in which the word GAIN appears once instead of twice); second, with the APPLY operator (doing all the PARMS without explicity mentioning them). Now imagine the savings with perhaps 20 parameters.

CONCLUSION

Many possible applications of this technique come to mind. One of them is the control of a number of actuators in a system (valves, shutters, etc.). They are usually best assigned Forth words for individual control, but one often wishes to close all of them with a single command. Another is a set of colon definitions that all must be done at some point, such as pass/fail tests. One can define a "class-smart :"as follows:

```
: :TEST : TESTS >CLASS ;
```

This automatically collects :TEST words in the class TEST . To perform all tests, just apply EXECUTE to TEST . Yet another application is in a system simulator, where each state variable (flip-flop state in a logic simulator, position or velocity in a dynamic simulator) is a Forth word. Every timestep, the "next value" of each state variable is computed from the "current value" of the variables. By putting all these state words in a class, one can apply an "update" operator (which shifts the contents of the "next" to "current" field in a state) to all state variables, without having a state array as such.

The use of classes and APPLY shows how powerful the general principle of "factoring out" redundancy in programs can be. In the case of APPLY , a control structure has been factored out, and the resulting code is not only more concise, but clearer: APPLY explains what is going on without the clutter of the loop construct. Extensibility leads to higher levels of abstraction embodied in Forth words. Abstraction, correctly applied (namely: with well-chosen names, good documentation, and a dedication to simplicity), is the ultimate tool for maximizing readability.