85-540

сообщения
объединенного
института
ядерных
исследований
дубна

Ц 840в

7033/85

E10-85-540

M.Winde

EXPERIMENT AUTOMATION DEMANDS FOR FLEXIBLE PROGRAMS

1985

1.0. FLEXIBILITY OF PROGRAMS FOR SCIENTIFIC EXPERIMENTS

The typical job of a programmer involved in a team of scientific experimenters is to write very special programs, i.e., programs which are needed only for one user group at one experimental installation for a limited time. Obviously the most important thing then is to have the programming done with minimal human effort. Other program quality attributes as efficiency, portability, ease of exploitation, etc., become less crucial.

The common method to achieve such programs quickly is to use a program system that provides many of the facilities commonly necessary. There is a number of such systems for experiment automation/1,2/.

Another method is to make use of libraries - special libraries for experiment automation and general scientific libraries are available.

The users are very satisfied, when first working with a program system suited for their tasks. They will spend much less time in programming, because they can concentrate themselves on the problem proper instead of concentrating on structural details of the program.

But in contrast to programs in many fields of application, programs for scientific experiments do "live", i.e., the program that was constructed at the beginning, will not be the final one because the original defined problem is not the final one. Instead of that problems become other ones step by step, so programs become other ones step-like too.

We will call that "the problem is open" due to uncomplete problem analysis in advance. Openess of problems is very typical for programs for scientific experiments.

Thus after some time programming usually tends to become more difficult. As a matter of fact this is often not only due to the problems that become a little more complex but also due to the program system that becomes much worse suited to the new problems. So after some time of unsatisfying struggle with the system the programmers will probably try the second way. The programmers now construct their program without using a program system. Instead, they write a special program very well suited to the problem to be solved. They make maximal use of existing (possible very good, complete) libraries.

The job is satisfying, once again. Because most of the program modules needed were already written to run under the program system, they may just be taken and linked. Typically only a small main program is needed to call them in a sequence. It is very easy to overcome all the inflexibilities and inefficiencies of the program system.

The program obtained this way is not the final one, of course. Modifications have to be made again. Some new options have to be included to make the use of the program more variable. After some time of exploitation and development the program becomes both hard to use and hard to modify.

It becomes hard to use because the options where incorporated "by necessity", i.e., the "syntax" of the "command language" of the program is non-uniform. It becomes hard to modify because its original structure has been likely destroyed by all those modifications.

What is the way out of this dilemma now? It is surely the construction of another program system. Now we know better where the limitation of its applicability will arise, so we try to construct a more flexible program system.

2.0. APPLICABILITY AND MODIFIABILITY

Every program is composed of a number of different modules. Some modules perform very special tasks, useful only in one program. Other modules perform general tasks. It suggests itself to construct these modules in a way to make them applicable in as much programs as possible.

Let us consider the problem by discussing an example, to begin with. The following module (written in a fictive language) calculates and outputs the squares of the first 100 positive integers:

```
   for i=1...100 do output(i*i); enddo;
```

If the task is to output the squares of integers 50 to 200 the module must be modified, i.e., the source text must be changed, then the program must be recompiled. Obviously the range of applicability of the module is very low. It becomes higher if the module asks for the lower and upper limits of the loop:

```
              ask 'lower limit': low;
              ask 'upper limit': up;
              for i=low..up do output(i*i); enddo;
```

The module is now applicable to calculate the squares of arbitrary integers without modifications. On the other hand input of lower limit and upper limit is necessary every time the module is executed.

The applicability could be increased further if other functions were allowed, instead of squaring. Simultaneously the expense in executing the program would increase - the function

has to be input. Besides this the expence of programming increases, the program becomes longer and more complex.

Hence, it is not practicable to increase the range of applicability of the module until "infinity" (a very general case of the module would be a formula interpreter, e.g.). The programmer rather must decide how general the program has to be, which of the "parameters" (in the most general sence) has to be variable and which may be considered as being constant. He has to take into consideration the possibility that he goes wrong in some decisions, i.e., a "constant" turns out to be a "variable" or vice versa. Hence, he has to work with the parameters in a way providing easy transition from "constants" to "variables" (and vice versa).

Thus the flexibility of programs and program modules is given by two factors:

1. Applicability: a program module is as widely applicable as it is possible to change its inner control flow by some input data in order to meet conditions not well defined or unforeseen at the beginning of programming.

2. Modifiability: a program or a program module is as much modifiable as it is easy to change it by editing its source text in order to accomplish unforeseen tasks. Modifiability of a program includes the exchangeability of the modules from which it is built.

3.0. FLEXIBILITY AS A CHARACTERISTIC OF PROGRAM QUALITY

In/3/ software quality is treated. It starts with defining some complex characteristics. Then going into detail, a number of basic characteristics are found establishing those more complex characteristics. In fig.1 the characteristics and their dependencies are shown. They are re-arranged from/3/ in order to better reflect the newly introduced complex characteristic "flexibility".

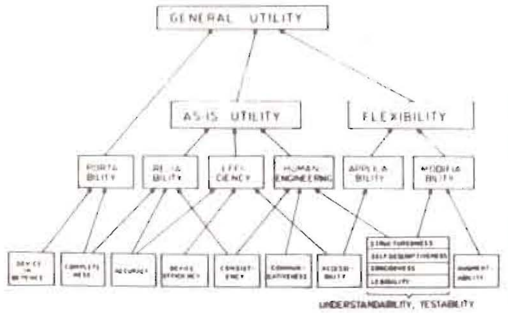

Fig.1. Characteristics of Software Quality.

We see that a program is flexible if it can be variably applicated, if it is well structured, selfdescriptive, if it easily accomodates expansions in data requirements or component computational functions.

Hence, all known methods to achieve these characteristics can be used. Most worth mentioning is the method of structuted programming. The programming language used also has significant influence on the flexibility of the programs.

After having defined what flexibility means we can go back to the original problem. How can we make programs flexible enough to live with for a longer period of time? As we have seen the conventional program system is not the best solution. Also a simple module library does not satisfy. What we need instead of that is a library of general modules which are easy to modify and some design principals to make variable and modifiable programs out of them.

Let us consider the problem of modifiable libraries to be solved (at least it can be solved with the help of well known methods). So we can concentrate ourselves on the design principles for variable programs.

A program consists of modules connected one with the other by a call-structure. The call-structure and a part of the input data determine the actual program control flow. Every module determines its successor, thus the program control is distributed among the modules.

Due to the already mentioned openess of the problem the actual program control flow heavily depends on the input data. It is usually influenced by decisions of the experimenter.

In very simple programs this is done by aborting the running program, modifying it, then re-compiling and re-running it. Already taken data must be lost.

To allow prompter reaction, there should be a possibility of controlling the program flow with the help of commands from the terminal console. For that purpose some modules must be able to accept commands. In the most simple way this is done by a sequence of write and read statements in the language used to program the module. Somewhat more convenient is the use of a dialogue package. A dialogue package guaranties on a certain level a uniform command syntax. Besides that some comfort is offered to the user (parameter prompting, parameter checking, repeated prompting in the case of illegal values, defaults). The actual program control flow is not effected by the use of a dialogue package. It stays distributed among the modules.

If centralizing not only the dialogue but also essential parts of the program control we get a loose program system, sometimes called a "software framework"/4/. And this seems to be exactly what we need, from the point of view of modifiability.

In a program system the user written modules merely offer some possibilities rather than fixing the program control. The actual sequence of execution of modules, frequency of execution, etc., is defined by the commands. Consequently it must be possible to define commands and to logically link them to modules, i.e., input of a certain command should activate a certain module.

```
                  SUBROUTINE HIST (IPAR)
                COMMON... IPASS, LUNDIS, LUNLPT
                INTEGER TITLE (10), BW
                GO TO (1000, 2000, 3000)
C
C*= PASS 1       COMMAND DEFINITION
C-----------------------------------------------------
1000            CONTINUE
                CALJ DECLR(HIST, 'define',5,1)
                CALL DECLR(HIST, 'show',1,2)
                                   ←— CALL DECLR(HIST,'print',1,2)
                RETURN
C
C*= PASS 2       COMMAND EXECUTION
C-----------------------------------------------------
2000            CONTINUE
                GO TO (2100,2200)IPAR                 ←— ,2300
C define
2100            CALL GETI('hist-nb.',NRH)
                CALL GETC('title', TITLE, 10)
                CALL GETI('nb. of bins', NRBIN)
                CALL GETI('lower edge', LOW)
                CALL GETI('bin width', BW)
                CALL BOOKH(NRH, TITLE, NRBIN, LOW, BW)
                RETURN
C  show
2200            LUN = LUNDIS
2210            CALL GETI('hist-nb.', NRH)
                CALL SHOWN(NRH, LUN)
                RETURN
C print                                         ←—
2300            LUN = LUNLPT                    ←—
                GO TO 2210                      ←—
C
C*= PASS 3       ACTIONS
C-----------------------------------------------------
3000            CONTINUE
                ... fill the histograms
                ...
                END
```

Fig.2. Modul HIST (in FORTRAN).

See fig.2 for example. A module is to be programmed for simple histogram manipulation. Input of a command DEFINE will define a new histogram, SHOW will present a histogram on a display screen. The module is composed of three passes. In pass 1 all the commands to be logically linked to that module must be declared. This is done by the CALL DECLR statement. Parameters are (1) the name of the module, (2) the command identifier, (3) the number of parameters and (4) the sequential number of the command (counted inside of this module). The first pass has to be executed once after program start.

The second pass has to be executed every time one of the commands logically linked to the module was input. After having identified the command the frame calls the module with parameters IPASS=2 and IPAR equal to the sequential number of the command. Control branches to label 2100 or 2200, respectively. Then the parameters have to be input. Finally the module calls subroutine HBOOK that will book the histogram (one of the various histogram packages available is used).

A third pass is necessary, of course, to fill data into the histograms. But this is not interesting from our point of view.

The module provides manipulation of an arbitrary number of histograms of different size, contents, etc.

If the modules are constructed prospectively and carefully, then not only many "through-away" modules come into being but also some modules usable in other applications. A command library arises in the course of time.

In general the modules are not ideally designed from the beginning. They have to be refined. In our example it turns out that histograms have to be presented on a printer too. We must add the command PRINT. This is a triffle. We have to declare another command in pass 1 and to code its execution in pass 2 (see fig.2 right side). Besides that we have to ensure only that the command is not defined elsewhere already. Updating any lists elsewhere in other program parts is not necessary.

This way it has to be. Declaration of additional commands must be easy to perform.

The method used to program module HISP was suggested by the class concept of languages like Simula or Concurent Pascal. In contrast to these languages the instantation of a class is not initiated by the program text but by commands via the terminal.

The formal translation of the class concept into a FORTRAN subroutine goes along with some insufficiencies.

1. There are some identical statements obligatory in all modules (COMMON, branches to the passes, parameter IPAR in the subroutine declaration). These compulsory statements reflect the standard of interaction of the modules in the program system. In the process of coding they are burdensome only.

2. The introduction of the "sequential number of the command" is obviously more pragmatic then elegant.

3. The passes of the module have to be executed at different times: pass 1 must be executed once after program start, pass 2 every time a command was input, pass 3 every time a value is integrated into a histogram. If the program runs on a computer with small main memory, we would like to define an overlay structure and to distribute the passes to different overlay segments. This is not practicable if all three passes are loca-

```
+ action HIST

+ command define
 NRH: hist-nb.:                int
TITLE: title:                 char (10)
NRBIN: nb. of bins:            int
LOW: lower edge :              int
BW:  bin width:                int

+command show; NRH::
+command print; NRH ::
+pass2
define: CALL BOOKH(NRH, TITLE, NRBIN, LOW, BW)
        RETURN
show:   LUN = LUNDIS
     10 CALL SHOWN(NRH, LUN)
        RETURN

print:  LUN = LUNLPT
        GO TO 10

+pass 3
        ... fill the histograms
        ...

+end
```

Fig.3. Module HIST (in a fictive language).

ted in one FORTRAN subroutine. Alternatively we can code the three passes in three subroutines. Unfortunately we will lose the unity of function and form, this way. Programming and modifying become harder to survey.

We can try to overcome these insufficiencies by the use of a precompiler/5/. The module may be input to the precompiler in a special language suited to the problem. Example module HIST thus obtains an easy to survey form (fig.3). The precompiler performs the translation into the form wanted by the program system. It translates the program text from fig.3 to the FORTRAN subroutine of fig.2 if there is sufficient main memory at the target computer or into three single subroutines if not, respectively.

On the other hand using a precompiler brings along some unforeseen side effects if a module is more complex, so it cannot be recommended in general.

4.0. PROGRAM SYSTEM AND PROGRAMMING METHODOLOGY

We discussed general problems of controlling programs for experiments by the commands. By introduction of simple standards we got solutions on a comparatively low level of integration. There is no uniform programming methodology supported by the program system. Instead, we have a collection of methods, standards to be followed, programming languages and commands. All that must be learned by the user. In detail it is necessary to learn the following steps when writing and using a program:

1. design: design methodology, possibly a design language;
2. coding: general programming language, system calls, standards, if a precompiler is used: special language;
3. Linking: command language of the linker program of the operating system, standards for linking the user modules with the frame;
4. execution: dialogue package, "command language" of the program;
5. test: command language of the debugger.

To perform all these steps knowledge of an editor is necessary too. To go from one step to another also minimal knowledge of the command language of the operating system is necessary.

The multiplicity of these demands to the user is a substantial barrier. A new user has to learn too much and makes too many errors until he feels some progress in programming efficiency at last. Hence the system should be developed in order to reduce this barrier. One way is the unification of all the languages mentioned above. There should be a programming methodology that is supported by the language to be constructed.

Thus the way we have to go is somewhat analogue to the current tendency in the field of data base systems. While in older data base systems the user modules were written in a general programming language thus demanding all the steps mentioned above, modern data base systems integrate one or even more languages to perform them uniformly.

The tendency to integrated software tools is also observable in other fields, especially in the field of personal computer application. In the field of experiment automation this way is gone by FORTH-applications/6/. Another possibility is presented in/7/.

5.0. SUMMARY

Software frameworks (loose program systems) seem to provide a suited structure for flexible programs. The program flow must be controlled by commands. Command modules (command driven classes) lead to a modifiable set of commands.

The multiplicity of demands to the user of a software framework has to be overcome by the construction of integrated software tools providing a uniform language for all steps of programming.

The author wishes to express thanks to V.J.Zhiltsov and B.G.Shchinov for some useful discussions.

REFERENCES

1. OC-DEC On-Line Data Acquisition Manual. CERN, Geneva, 1982.
2. Bartelett J.F. et al. Fermilab MULTI User's Guide, FNAL, PN-97.5, 1979.
3. Boehm B.W. et al. Characteristics of Software Qaulity. North-Holland Publ.Co., Amsterdam - New York - Oxford, 1978.
4. Williams G. Software Frameworks. Byte vol.9(13), 124-127, 394-410, December, 1984.
5. Pfeiffer G. A Flexible Command Generation Technique for Application in Diagolue Systems. Software - Practice and Experience, vol.14(5), 484-489, May, 1984.
6. Hogan T. Discover FORTH. Learning and Programming the FORTH Language, Osborn/ Mc.Graw-Hill, Berkeley, 1981.
7. Schenk J., Wegner P., Winde M. JINR,E10-85-541, Dubna, 1985.

Received by Publishing Department
on July 12,1985.

Винде М.   Е10-85-540
Автоматизация экспериментов требует гибкие программы

При разработке программ для научных экспериментов, особенно для on-line сбора данных, как правило невозможно предусмотреть все детали заранее. Поэтому программы должны быть гибкими, т.е. допускающими настройку на различные условия, и легко модифицируемыми. В статье вводится понятие гибкости как характеристика качества программ. Обсуждаются некоторые общие методы достижения гибкости программного обеспечения. Показано, что "эскизное программирование" /не полностью определенные программы/, может обеспечить структуру, дающую необходимую гибкость. Реализация команд в виде модулей обеспечивает модифицируемость набора команд. Показана необходимость создания "интегрированного программного инструмента" для обеспечения сбора данных в эксперименте, который должен обеспечивать унифицированный язык для всех этапов программирования и применения программ пользователя.

Работа выполнена в Отделе новых методов ускорения ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна 1985

Winde M.   E10-85-540
Experiment Automation Demands for Flexible Programs

When writing programs for scientific experiments, especially for on-line data acquisition, it is usually not practicable to complete the detailed problem analysis in advance. This demands for flexible programs which can be variably applicated and easely modified. The paper introduces flexibility as characteristic of program quality. Some general methods to achieve flexible programs are discussed. Software frameworks (loose program systems) seem to provide a suited structure for flexible programs. Command modules (command driven classes) lead to a modifiable set of commands. The necessity to construct an integrated software tool for on-line experiment data acquisition is shown. It should provide a uniform language for all steps of programming and applicating of the user program.

The investigation has been performed at the Department of New Acceleration Methods, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna 1985