On the Necessary Evolution Towards Improvement Specialization in Software Production Teams

J. Bézivin
F. Gauduel
J.L. Nebut
R. Rannou

I.R.I.S.A.
Université de Rennes
B.P. 25A
35031 RENNES Cedex
France

1. INTRODUCTION.

Human aspects are an essential element in the software production cycle. This has been widely recognized and several proposals such as the "chief programmer team" have proved their effectiveness as methodologies to produce high quality software at a lower cost. In this paper we investigate the possible short term implications of recent technological and methodological evolution on the basic structure of software production groups.

As diversity of the underlying architectures increases (e.g. multi microcomputer systems), programming languages are becoming more and more independant from the particularities of these machines (e.g. new specification languages). At the same time, advocated by several people from different horizons (e.g. (BAUER,1976), (KNUTH, 1974), (WEINBERG,1972)), a new programming methodology seems to become credible. It consists in producing first a correct program without any concern for performance. The program is then iteratively transformed into a logically equivalent but more efficient version. Methods to support these transformations range from very theoretical flow analysis algorithms to more pragmatic techniques. As part of a research project on system performance improvement, we have been led to consider that an ideal system writing language should permit to express on one side qualitative specifications for the system, and on the other side quantitative directives intended to improve the behaviour of the system in a given environment. The feasibility of the methodology has already been par-

This research is supported by CNRS under grant A.T.P. 2317.

tially proved by the LIS language (ICHBIAH,1975). After having recalled the major characteristics of the approach, we discuss in this paper its social and professional implications. It is particularly argued that a lot has to be gained from the separation of software production in two phases (qualitative specification ; improvement) and from the assignment of different people to these tasks.

2. TRENDS IN THE SOFTWARE PRODUCTION PROCESS.

Several shifts in emphasis have occurred in the short evolution of software development methodology. In the first ages of programming, efficiency was the main concern. Slowly people began to become convinced that in addition to direct operating costs (memory and processor consumption), a second class of costs (related not with hardware resources but with staffing) was of paramount importance. Among these "non operating costs", one may quote programming costs, documentation costs, modification costs, maintenance costs, etc ... New methodological proposals that can be described under the generic name of "structured programming" soon became credible. They are characterized by a major emphasis put on the logical structure of programs.

Reactions from the programming community to these proposals (which were at the beginning mainly academic proposals) have not always been favourable. A paper by KNUTH (KNUTH,1974) expresses a very balanced view of pro and cons of Structured Programming. This article presents methodological aspects of program construction with special emphasis on performance problems. One main point put forward by KNUTH is the following :
"... *premature emphasis on efficiency is a big mistake which may well be the source of most programming complexity and grief. We should ordinarily keep efficiency considerations in the background when we formulate our programs ... And when it is desirable to sacrifice clarity for efficiency ... it is possible to produce reliable programs that can be maintained over a period of time, if we start with a well structured program and then use well understood transformations that can be applied mechanically. We shouldn't attempt to understand the resulting program as it appears in its final form, it should be thought as the result of the original program followed by specified transformations. We can envision program manipulation systems which will facilitate making and documenting these transformations ...*"

We think that this view expressed by KNUTH may well become an essential characteristic of the forthcoming software production methods and tools. Several remarks can be made to assertain this opinion.

Premature emphasis on efficiency problems obviously bears a lot of drawbacks. It diverts energy from the logical design and thus in-

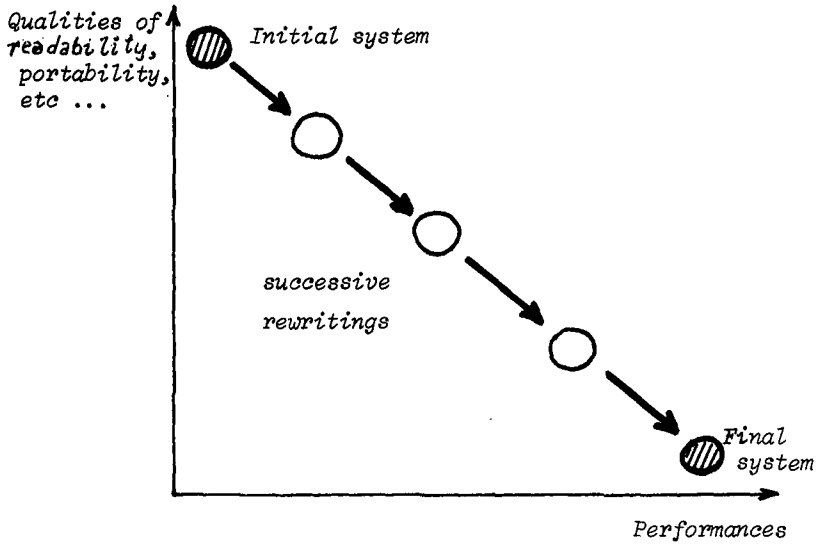


Figure 1.

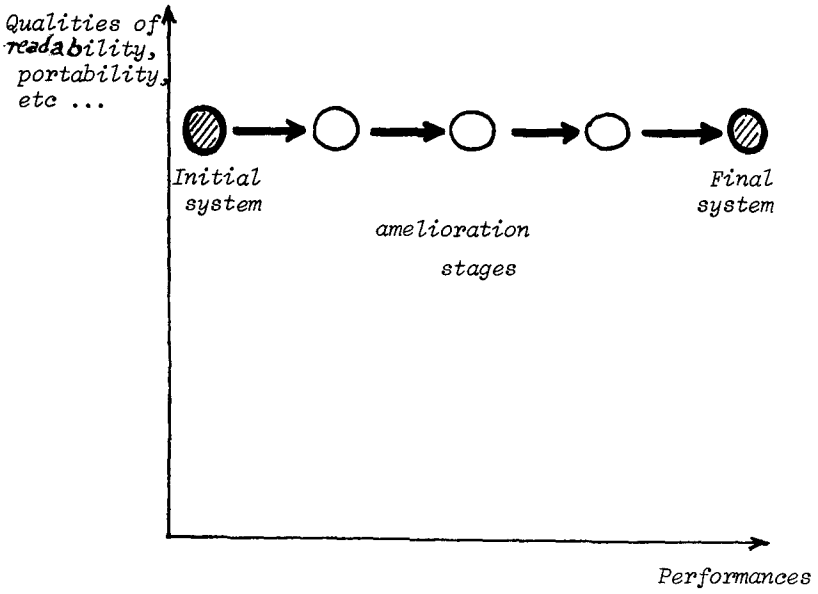


Figure 2.

creases the risk of producing unclear and therefore unreliable programs, putting up the indirect costs of debugging, maintenance, modification, etc ... This has been clearly established by WEINBERG and SCHULMAN (WEINBERG,1972). They conducted a series of experiments to explore several aspects of programming performance. For example they asked five groups of programmers to do the same program but gave them different goals : minimum core, maximum output clarity, maximum program clarity, minimum statements, minimum production time. One interesting result is that each group ranked first on its own objective. But other conclusions were also drawn from the experiment and one of them is interesting for us. The groups with efficiency objectives (minimum core and minimum statements) ranked fourth and third on program clarity. In fact the experiment showed that some objectives tend to be highly correlated (e.g. program clarity and output clarity) and some other are highly conflicting like efficiency and program clarity.

Another major drawback of premature emphasis on efficiency is related to portability. As a matter of fact it is hard to define what is really meant by efficiency, when designing a system. Efficiency is meaningful only when it is related to a particular environment. Environment may be viewed as composed of two parts : the user context and the machine context. By machine context, we mean all that has to do with the hardware supporting the system. By user context, we mean the system load. Many efficiency choices are taken on the grounds of an assumed characteristic of the environment. Thus what is an immediate optimisation may turn out to be a pessimisation when the system is transported to another site.

A lot of arguments can thus be found to support the view that production effort must first be spent on logical design and only when this phase is achieved on performance improvement. Unhappily even a posteriori enhancement of programs may have negative effects on clarity and readability. It is often done by finding tricks and simplifications that permits to gain some words of storage or some milliseconds of execution time. This gain is often swept out by the loss induced by increased maintenance costs. To summarize, figure 1 illustrates the usual coding-improving scheme whereas figure 2 illustrates an approach we would prefer.

As part of a research project on program performance improvements, we have been led to wonder under which conditions an ideal approach such as the one illustrated by fig. 2 - enhancement without loss of readability - could take place. The answer we have come with is that what we need is some kind of two level language. One level will permit to express the logical structure of the problem and the second will serve to describe optimisation actions on the first program. The remainder of this section illustrates the fact that such an approach has received attention and is already partially implemented in several places.

One of the most significant example of two-level specifications is given by the LIS language (ICHBIAH,1975). As a matter of fact, the algorithm is written using very high level language features. Then, an implementation part allows the programmer to choose particular implementations of the data of his program, according to a specific physical organization ; a low level language is also provided in order to be able to get access to the machine characteristics, including the notion of interface which enables the programmer to use machine instructions and to link together LIS and non-LIS programs.

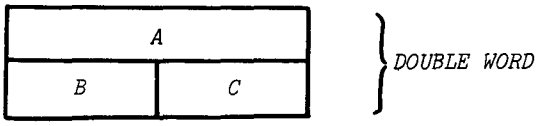
In the following example :

```
segment data MAIN
        :
        type T = plex
                    A : integer ;
                    B : boolean ;
                    C : boolean ;
                end ;
          :
          :
        P : action (X : integer, Y : out integer) ;
          :
          :
end
```

```
implementation
        :
        :
     type T = plex
                 DOUBLE ALIGNED ;
                 A : word (0 : 1) ;
                 B : half (0 : 1) of word 1 ;
                 C : half (1 : 1) of word 1 ;
              end
        :
        :
     interface LINK_CONVENTION ;
            X : in R3
            Y : out R4 ;
            use R1, R2 ;
            begin
                 BAL,R1   LINK_CONVENTION
            end ;

     for P use LINK_CONVENTION
          :
          :
end
```

T will be implemented as follows :



The automatic data structure selection system, described by LOW (LOW,1974), shows us another model of a two-level approach ; this system chooses low level implementations for abstract information structures, among a fixed set of possible representations, in order to minimize the total time-space product. The system is used on programs where data structures are expressed in terms of such high-level information structures as sets, sequences and relations (programming language SAIL) ; this data structuration leads to logically clear and well structured programs which are designed and debugged more quickly (It is a general remark anyway that the higher the abstraction level of the language is, the easier it is to find effective improvement techniques (BAUER,1976). For choosing appropriate implementations, the selection phase needs information not only about the different representations available (storage cost function for the representation and individual execution time function for the primitive operations), but also about the use of the abstract data structures of the user's program (that information is obtained by static analysis, monitoring the execution of the program (using default representations) and by asking information from the user during an interactive interrogation phase).

A semi-automatic data-structuring method presented in (SCHONBERG, 1977), for a similar set oriented language, SETL, is based on the same principle ; the efficient data-structuring of a SETL program is obtained by supplying the language processor with detailed declarations of structural relations related to the program variables. In the absence of user-supplied declarations, the SETL processor chooses for the variables, a default representation which is reasonably efficient for the primitive operations which appear most frequently.

For example, one can write :

```
        :
(1)  repr S : set (ε B1), ε B2 ;
        :
(2)  repr X : ε B1, ε B2  ε B3 ;
        :
```

where (1) specifies that S is a subset of B_1 which is also to be considered as an element of the second base B_2
(2) indicates that the object X is to have three simultaneous representations, as element of each of the base sets B_1, B_2, B_3.

In numerous additionnal program improvement systems, the "two-stage programming" style assumes different aspects ; we may point out the system described by Burstall and Darlington in (BURSTALL, 1977), for transforming programs which are expressed as recursion equations into iterative and more efficient versions. This system is based on so called transformations rules and relies on guidance from the user. The compilation model advocated by Loveman in (LOVEMAN,1977) falls in the same category too. It is founded on the use of source to source transformations performed on source language representations of a program.

In this section we have shown that a noticeable evolution is taking place in the field of programming methodology. More and more proposals are oriented towards separation of the production process into several different phases and particularly towards separation of the design/coding phase from the improvement phase. Effective tools to support this kind of methodology are beginning to appear and usually take the form of a two-level language : one level for qualitative specification and the other for quantitative improvement. In that respect, D. LOVEMAN writes :
"... *The approach we favor, however, is to allow the programmer to be the strategist and to provide a mechanical assistant to perform the optimisation itself. ... Finally, a programmer is free to use all the facilities of his programming language to produce a high level, well-structured, modular program with the knowledge that should it prove to be inefficient, he has access to a set of tools to change representations and eliminate modularization overhead while perserving the correctness of his original program. ...*"
(LOVEMAN,1977)
It is therefore important to evaluate the possible implications of this evolution upon the production group structure.

3. TOWARDS PLURIPROGRAMMING.

It seems to us that division in methods and tools naturally induces the same division in personal profiles for the production people. This means that in addition to the usual "algorithm programmer" we shall see the arrival of a new intervenant in the production cycle : the "improvement programmer".

One first point that must be stressed is that this situation is not entirely new. The role of the system programmer in many computing centers is usually to take a constructor made system and to adapt it to some given environment. Tools available to him are for the most part of a rudimentary nature (system generation tools)

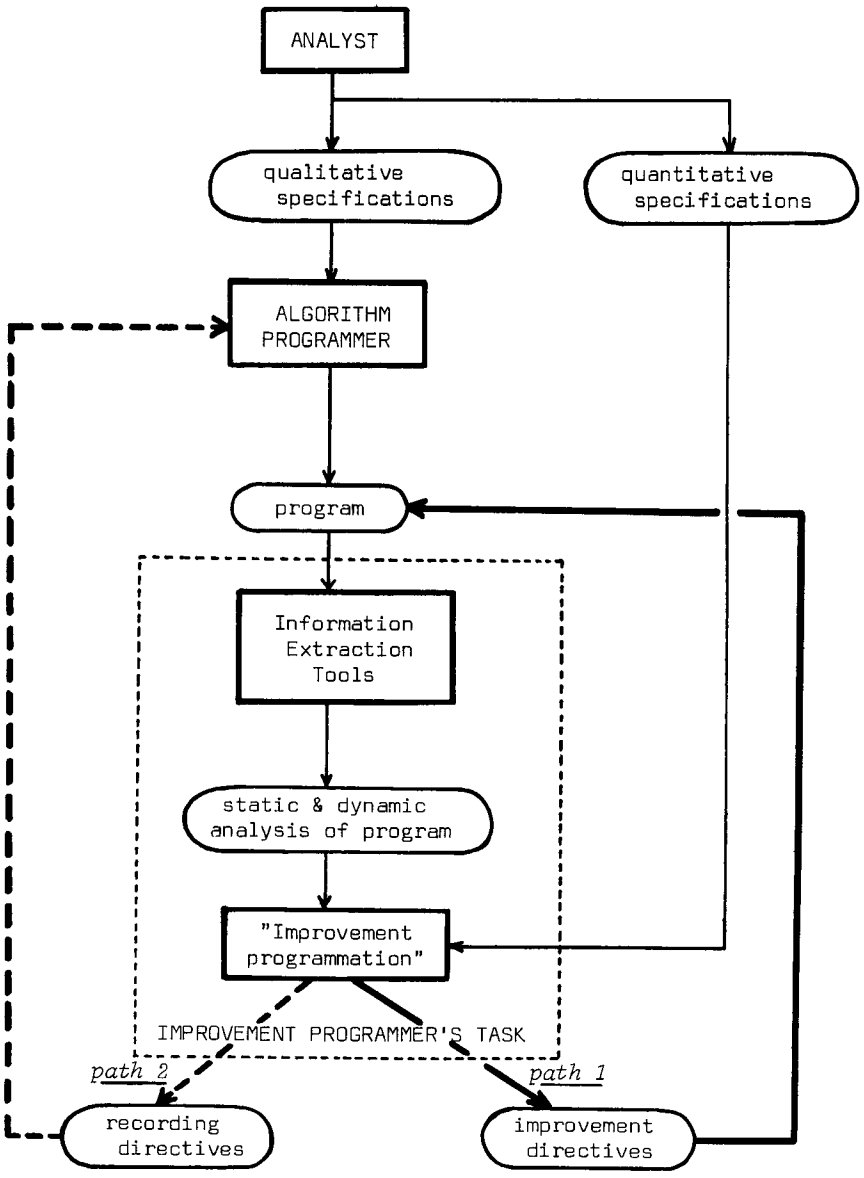


Figure 3.: *Overall relations between ANALYST, ALGORITHM PROGRAMMER and IMPROVEMENT PROGRAMMER*

but his work is essentially an improvement programmer's work.

We shall give in the remainder of this section some details on the work of the improvement programmer as we see it. Such a prospective description cannot be very precise because it is highly dependant on the programming tools that will be available in some years. Nevertheless we hope it is sufficient to convey the idea that the optimisation task is distinct enough from the programming task so as to justify the assignment of different people to these different works. As noticed by B. WEIGBREIT (WEIGBREIT,1975), the quantitative analysis of many algorithms requires considerable mathematical expertise. In our opinion, this required expertise goes beyond what one can honestly expect from an average application programmer. To put it in other words the monumental work of KNUTH dealing with quantitative evaluation of programs (KNUTH, 1968) for example, cannot be entirely assumed in a normal computer science curriculum. It represents however only the emerged part of the iceberg in the huge study domain of performance evaluation and improvement.

We have drawn in fig. 3 a simplified illustration of what the relations between the analyst, the algorithm programmer and the improvement programmer could look like. Although this is only a naive picture intended to clear up our mind, it will help us define what the work of an improvement programmer could look like.

In this simplified graph, the algorithm programmer produces a program compatible with the qualitative specifications provided by the analyst. This program is then processed by automatic tools. (The error correction process takes place at this stage but has not been represented in the graph because we are not concerned with this problem here). The improvement programmer's work starts at this point and his intervention will take three forms :

(T1) Extracting the relevant information from the "program".

(T2) Checking that the characteristics of the program do not violate the quantitative requirements set up by the analyst. (If they do, then taking the necessary actions).

(T3) Trying to reduce as much as possible the resource consumption of the program.

The improvement programmer performs thus a double task : performance analysis and improvement control. The latter means taking reaction decisions that may take one of the two following forms :

(R1) By using the facilities offered to him in the second level of the language, the improvement programmer will write a sequence of improvement directives. (*path 1* in the graph).

(R2) Or he may forward to the algorithm programmer a demand to rewrite some parts of the program according to some more precise guidelines. (*path 2* in the graph).

The second path is absolutely necessary because ... *"there is obviously a limit to how far a compiler"* [or a program improvement system] *"however sophisticated, can eliminate the effects of bad programming"* (BATES,1976).
This fact has been widely recognized by those who are involved in the area of program improvement transformation. We can quote B. WEGBREIT for example :
"... While the techniques we have presented can yield some interesting results, it would be a mistake to overestimate their capabilities. They are limited in effect to the transformation of one program to a better one. Case in which the input/output mapping can be better realized by a radically different algorithm are beyond the scope of this method. For example we can see no way to transform a definition of bubble-sort to a version of quicksort..." (WEGBREIT,1976).
The usual behaviour however for the improvement programmer would be to use possibilities (R1) and (R2) in that order. Hopefully in most cases, when the available tools will be rich enough, possibility (R1) will be sufficient to meet the objectives.

As we have seen it, the work of the improvement programmer will thus have the following characteristics :
i) A main part of it is related to information gathering. In order to do this, many classical methods and tools are available. They are related either to the analysis of the program ifself (static analysis (LOW,1974), assertions or relations declared by the user, ...), or to the study of the program within its execution environment (hardware or software monitoring, execution profile techniques (KNUTH,1971), (INGALLS,1971),...). In the same way, the knowledge of execution environment features (particular hardware mechanisms, instruction repertory, computing speed, ...) may supply useful information.
ii) It is essentially iterative in nature. That means that once an optimisation action has been taken, the system has to be evaluated again in order to characterize its new behaviour and to validate the effect of the optimisation.
iii) It is moderately interactive. A set of tools will be available to help the improvement programmer in his task and will be available in a semi-interactive way.
iv) There will be important interactions with the algorithm programmer. As we said earlier if a program has been very badly designed there is little hope to transform it in a good one by successive optimisation. In that case the improvement programmer will play an important educative rule towards the algorithm programmer by handling him back new programming directives. The objective is that this latter will learn in practice to produce very clear programs with acceptable performances. Such a program is likely to be improved by a significant amount in the optimisation phase.

4. CONCLUSION.

Classical methods of software production have proved *de facto* their applicability but have severe limitations (GOLDBERG,1973). New proposals to produce at a lower cost software of better quality are hampered by the rigid frame into which the same people are asked to deal with several different production tasks. Nevertheless, an evolution towards separation of the production process in two phases (qualitative construction, then quantitative enhancement) can be noticed in that respect among language designers.

In this paper, we have tried to show how this separation and assignement of different people to different tasks seem to us an ineluctable evolution and a good move. The evolution seems ineluctable because :

(i) It is no more possible to rely only upon optimizing compilers which however clever they are cannot take into account all the possible improvement transformations.

(ii) It is no more realistic to add new integrated improvement mechanisms to programming languages (e.g. the *packed* attribute of the PASCAL language). We have described elsewhere (ANDRE,1977) the pollution-like problem faced by programming languages that are being forced into accepting and integrating a huge variety of different mechanisms related not only to improvement but also to correctness proofs, error handling, tracing, documentation, ...

The evolution seems to bear many potential benefits as one may judge from the first results of the experiments conducted in that domain (the LIS language is already available within an industrial environment and used to write operating systems).

It is our opinion that if the programming team structure is flexible enough to accomodate the improvement specialisation move, then the optimisation methods and tools that are becoming available will be of paramount importance in tomorrow's software production environment.

REFERENCES

(ANDRE,1977) ANDRE, J. & al.
Professional needs for Software Morphology.
Presented at TC2 Conference on Constructing Quality Software, Novosibirsk, May 1977.

(BASILI,1975) BASILI, V.R. & TURNER, A.J.
Iterative Enhancement : A practical technique for software development.
First National Conference on Software Engineering, IEEE Comp. Society, Washington D.C., Sept. 1975.

(BATES,1976) BATES, D. (Ed.)
Program optimization.
Infotech State of The Art Report.

(BAUER,1976) BAUER, F.L.
Programming as an Evolutionary Process.
Invited Lecture, 2nd International Conference on Software Engineering, San Francisco, Oct. 1976.

(BURSTALL,1977) BURSTALL, R.M. & DARLINGTON, J.
A Transformation System for Developing Recursive Programs.
JACM, Vol. 24, N° 1, Jan. 1977.

(GØLDBERG,1973) GØLDBERG, J. (Ed.)
Proceedings of a Symposium on The High Cost of Software.
Stanford Research Institute, Sept. 1973.

(ICHBIAH,1975) ICHBIAH, J.D. & al.
The System Implementation Language LIS.
Technical Report 4549 E/FN, CII, 1975.

(INGALLS,1971) INGALLS, D.
The Execution Time Profile as a Programming Tool.
in Design and Optimization of Compilers, R. Rustin Ed., Prentice Hall Inc., 1971.

(KNUTH,1968) KNUTH, D.E.
The Art of Computer Programming.
Addison Wesley.

(KNUTH,1971) KNUTH, D.E.
An Empirical Study of FORTRAN Programs.
Software Practice & Experience, Vol. 1, N° 2, 1971.

(KNUTH,1974) KNUTH, D.E.
Structured Programming with go to Statements.
Computing Surveys, Vol. 6, N° 4, Déc. 1974.

(LØVEMAN,1977) LØVEMAN, D.B.
Programming Improvement by Source-to-Source Transformation.
JACM, Vol. 24, N° 1, Jan. 1977.

(LØW,1974) LØW, J.R.
Automatic coding : choice of data structures.
Technical Report n° 1, University of Rochester, Aug. 1974.

(SCHONBERG,1977) SCHONBERG, E. & LIU, S.C.
Manual and Automatic Data-structuring in SETL.
Ecole de l'IRIA : "Les langages de très haut niveau", May 1977.

(WEGBREIT,1975) WEGBREIT, B.
Mechanical Program Analysis.
CACM, Vol. 18, N° 9, Sept. 1975.

(WEINBERG,1972) WEINBERG, G.M.
The Psychology of Improved Programming Performance.
Datamation, pp. 82-85, Nov. 1972.