THE SOFTWARE CONFIGURATOR :
AN AID TO THE INDUSTRIAL PRODUCTION OF SOFTWARE

Dr. Federica Liguori

Syntax S.p.A.
Via G.Negri 8 - Milano (Italy)

Prof. Ing. Fabio A. Schreiber

Istituto di Elettronica-Politecnico
P.za L.Da Vinci 32 - Milano (Italy)

The different requirements of the main software producers - Manufacturers, Software Houses and Users - are briefly examined.
Then the Software House problems are considered in more detail as to the very different types of software it must produce and as to the necessity of fully utilising the internal know-how.

The collection and generalization of already produced software semi-manufactures and the diffusion of their knowledge through a very concise form of documentation, called 'Software Configurator', is presented as a way for the know-how exploitation.

Key words: documentation, know-how diffusion, methodology, program families, software components, software production.

Introduction

Problems related to software design and production are of outstanding importance as well for computer manufacturers as for software houses.
Since the NATO conference of 1968 (1) different themes have been recognized to be worth of research in Software Engineering.
Among them, the development of tools and methodologies for software design in order to assure the building of 'a priori' correct programs, which gave rise to the branch of software 'philosophy' called structured programming, the production of an easy readable yet complete documentation, the development of project management techniques specially suited for software work, work organization for software production, and others.
However we must observe that design problems have risen much more interest among researchers than production problems even if some design techniques and some tools are directly relevant to the production cycle; it is the case of the top-down approach, and of the development of high level programming languages.

Even when work organization has been considered, it was mainly in the environment of very large project management; it is the case of the 'Surgical team' or 'Chief programmer team' concept proposed by IBM (2) (3) (4).

In this paper we want to examine the problems of software design and production in a Software House, where many different projects are carried on simultaneously and where a continuous flow of offers and tenders making requires an extensive activity of gross analysis of new systems. The software configurator is proposed as a tool which can aid the system analyst both in the preliminary offer and in the actual design and implementation phases.

The Software House Problems

Traditionally, among software producers we can distinguish three classes: Computer Manufacturers, Software Houses, and Users.

Let us briefly identify their characteristics: we mean as a User a software producer employing no more than ten to twenty programmers on small-to-medium sized programs referring to problems peculiar to the user's application. This definition therefore excludes very small users who must rely upon large Services Bureaux or Software Houses, and it excludes very large users as well, since in this case the EDP department has the very same problems of a Software House.

We mean as a Manufacturer a software producer particularly involved in 'dressing' with system software a 'bare' machine.
This can result in very large projects, but, even if many people are involved in them, they have generally a unique manager responsible for coordinating all the activities in the project itself.
Again we exclude from the manufacturer's definition those activities which can be assimilated to those of a Software House (customers support, etc.).

The Manufacturer then is very interested in methodology issues, in reliability, in software distribution and maintenance, while the User is more concerned with portability, and extensibility of his application programs.

Software Houses have quite different problems, their activities ranging from system software, to applications, to dedicated special purpose systems. When the personnel involved in technical work exceeds a score, the diffusion of the know-how on the different projects in course of development or already terminated becomes very difficult.
So people tend to specialize on a rather narrow range of problems without even being interested in throwing a glance to what happens at the nearby desk.

However many are the projects, even if very different in nature and scope, which can benefit of identical pieces of software.
For example a string analyser routine can as well be suited for an assembler or a compiler as for the verification of the input in a conversational data collection system; a routine which computes the time elapsed between two dates can be useful to an inventory control system, a banking system, an operating system.

However since different people work at system software, at dedicated systems and at application systems, it is very likely that the same function be designed and implemented independently, so inventing the wheel again and again.
Aim of a Software House then, must be, among others, to avoid the production of even more new programs in a prototypal way, while making previous experiences widely known at its interior.

Moreover it is essential that this knowledge be available both to system designers and to marketing people.
In fact, when proposing a system to a customer or when replying to a tender, the knowledge of the availability of already implemented functions can lower noticeably the man-month count, so making the offer more convenient both from the economical and time-of-development points of view.
On the other hand, if the system designer is aware of the existence of pieces of software, which could be useful to him, at the very beginning of the design stage, he can insert them in the system by arranging in advance the proper interfaces and guiding the design choices, which otherwise are often a matter of taste.

Such a way of working will hopefully aid in producing less individualistic software which could be even more generalized.

The Software Configurator: Its Background

As mentioned in the previous section, the problem for a Software House is then twofold: at one side there is the need of knowing what can be reutilized from previous experiences, at the other side the need of having a pool of software modules, ranging from the small subroutine up to the package size, which can be used either directly, if the programming language is compatible with the new application, or which needs only to be coded again.

As to the second problem we found very interesting the works of Mc Ilroy on 'Software components' (5) and of Parnas on 'Program families' (6).

In the first of these works the production of software components (routines) is proposed in a completely independent production cycle as the software which may use them.
In the very same way that many different kinds of resistors, transistors, etc., are produced by component manufacturers and assembled into computer hardware or other electronic circuits by other people, software components performing specific functions, but with different levels of performance as to precision, time-memory requirements, reliability, etc., could be produced and used by different people, even by different manufacturers.

Mc Ilroy analises all the economical and marketing implication of an independent component subindustry, and even if there are many arguments which could make its profits doubtful, they do not apply if the components themselves are produced as a by-product of the normal production of a Software House.

However, to be really useful, software components or semi-manufactures must be rather general and possibly they must be available in several different versions.
The problem then consists in tracing back all the design decisions which led to an already available software component, designed and implemented in the context of some particular project, and to find how it could be made more general by identifying those design decisions which could give rise to an entire 'family' of programs and by 'hiding' them in perfectly compatible 'plug-in' modules (6) (7).

This 'bottom-up-top-down' approach, which starts from 'home available' components to build families of functional modules, obviously requires an extra effort and investment from the Software House, but it can be well worth to reduce the overall man-month count as far as the functions are encountered very frequently in different projects.

To this end a group of people is to be set-up, under the responsibility of the System Engineering Service manager, whose task is to evaluate the software produced under the aspect of its reusability and generalizability, and relying for their decisions on their experience and forecasts of the kind of projects the company will most probably develop in a next future.

When some interesting function is found, it is analysed and the specifications for the implementation of the relevant modules are passed to the Technical Service for realization.

However, while the general analysis in terms of information hiding modules can generate large program families, the members of which can differ as to the object computer, computation accuracy, physical implementation of data structures, etc., only some members will actually be produced as components, the others remaining as potentialities, in the sense that only the analysis and design phases will be accomplished, while actual coding and testing will not.

Therefore fundamental prerequisites for such an action are the following:

- the availability of a very skilled technical staff for the evaluation and the reanalysis of the produced software
- homogeneous development methodologies and documentation techniques at least within the different application sections of the company. Modular, besides structured, software production techniques are obviously useful tools in achieving the goal.

As to the first problem, that is the dissemination of the knowledge about the availability of software components, we decided to use a very concise form of documentation for each component whether it is actually implemented or it is only a potential member of an already analysed program family.
All the components file-cards are collected in a manual, which has been called the 'Software configurator'. This book is the tool which allows the project leaders and the system analysts to see at a first glance if something useful to them is already available.

To aid them in the search, three kinds of indexes are provided:

a) - alphabetical ordering for the <u>component names</u>

b) - alphabetical ordering for the <u>component functions</u>

c) - alphabetical ordering for the <u>hardware environments</u> in which the components operate with a sub ordering for <u>software environments</u> for the same hardware.

The file-cards are actually grouped following the functional ordering and the functional index is implemented as a thumb-index.
This choice has been made since the most common search criterion is the functional one.

A future goal is to store the Software Configurator in a computerized library allowing its use through a terminal and a set of information updating and retrieval on-line procedures.

As we mentioned in the previous section the Software Configurator is a sort of very concise documentation, yet giving all the information necessary in deciding wether it is worth-while to go deeper into the extended documentation of a certain component.

In this section we are going to present the framework of a file-card and the internal procedures to build and update the Software Configurator.
Let us examine then the file-card structure. We shall give the heading, followed by a short explication of its contents.

1.- <u>Name</u>

The name of the component is given both in the extended version and, if applicable, in the acronym form.

2.- <u>Producer</u>

The name of the company who produced the component. In a Software House infact very high is the probability of using external packages, which can nevertheless be considered as components.
If the component is home-made, the name of the project-leader under the responsibility of whom it was designed and implemented, is mentioned.

3.- <u>Function</u>

The function of the component is concisely explained.

4.- <u>Status</u>

The status of the component (actual or potential) is stated.
If it is an actual component it is specified in which form (source or object code) and on which supports (cards, tapes, cassettes, ...) it is available.

5.- <u>Environment</u>

5.1 <u>Hardware</u>

The hardware on which the component is running is described in terms of:

1 - CPU: manufacturer and model
2 - Central memory: system + user workspa ce
3 - Mass memories: type and amount
4 - Peripheral devices
5 - Special devices: (if applicable) memory management, floating point arithmetics, etc.
......

5.2 Software

The system software under which the component has been designed is described in terms of:

- operating system
- language processors
- file management system (access method)
- TP monitors
......

This is not an exhaustive list and the elements mentioned above are listed only when applicable.

6.- Aim

The aim for which the component was conceived in the original system:

- extension of existing system software
- optimization of time/memory constraints
- others.

7.- Installation

The component requirements are described in terms of:

- memory requirements for the object code
- memory requirements for tables
- memory requirements for work areas
- common areas

8.- Description

A very concise description of the component is made in terms of its structure, of the algorithms, and of the data structures it uses.

9.- User Interface

In this section it is explained how to use the component; which are its input and output parameters, which are the control parameters, etc.

10.- Documentation

All the available documentation on the component and its environment is listed.
The documentation itself is kept in the company's library.

When the decision of including a new component in the software configurator has been taken, the System Engineering Service prepares the file-card for it. Then the file-cards are sent to the project leaders, under the responsibility of whom the components have been implemented, for checking their correctness and completeness.

The final version is then distributed by SIS. The project leader retains the responsibility for informing the SIS of any variation or updating which has been made on an existing component.

Summary and Conclusion

The need of circulating the know-how about products developed by a Software House has led to the development of a particular form of documentation under the form of file-cards included in a 'Software Configurator'. In this handbook are included all the pieces of software, ranging from the complete package to the single subroutine, which can be considered as software components of very complex systems.

Different versions of the same functional component, obtained as families of programs, are kept for different applications.

Presently the Software Configurator is still in an experimental stage and it is possible that changes to the described framework will be made according to the feedback provided by analysts and project leaders. However the first reactions of these 'users' have been very favourable since they found it a useful tool and an aid in the early design stage.

In the Appendix one of the file cards is presented.

Bibliography

(1) P.Naur, B.Randell (Ed.) - 'Software Enginee - ring' - Report of the NATO Science Committee, Jan 1969
(2) H.Mills - 'Chief Programmer Teams, principle, and procedures', IBM report FSC 71-5108, Gaithersburg, Md., 1971
(3) F.T.Baker - 'Chief programmer team management of production programming', IBM Sys.Journal, 11,1 (1972)
(4) E.P.Brooks - 'The mythical man-month' - Addison-Wesley publishing Co., - 1975
(5) M.D.Mc Ilroy - 'Mass produced software components', in (1)
(6) D.L.Parnas - 'On the Design and Development of Program Families' IEE Transaction on Software Engineering, Vol. SE-2, nº 1, March 1976

(7) D.L. Parnas - 'A Technique for Software Module Specification with Examples' - Communication of the ACM, Vol. 15, N° 5, May 1972

Appendix

DATA HANDLING SYSTEMS

STRUCTURED DATA ENTRY FILE SDEF

PRODUCER : SYNTAX SpA - Mr. G. Verdi

FUNCTIONS

SDEF (Structured Data Entry File) allows the dynamic handling of more than one 'logical file' used as if it were only one file even with a multivolume organization.
This system is specially useful for 'Data Entry' like applications where different programs (JOB) have to handle more than one data file (BATCH).

STATUS

The first version of SDEF system is available which manages only single volume files. A second version which provides the management of multivolume files is available only for a limited set of functions supporting at most two volumes. The source programs of SDEF are available on tape or disk.

ENVIRONMENT

. Hardware	:	PDP11/35	with DKC40 disk unit
. Software	:	RSX11M v. 2	with File Control Service

AIM

The main purpose of SDEF is to increase the features of FCS as it provides the following functions:
- to perform logical deletes of records
- to process a file forward and backward
- to alternate updating and control phases

Moreover the use of SDEF allows the reduction of the memory space occupied by 'File Control Block' when more than one file must be contemporary opened.

INSTALLATION

SDEF runs in a user partition.
Data and coding, except the common working area, occupy 5 K bytes of main memory.

DESCRIPTION (Single volume version)

File Structure

The SDEF file is organized with pointers to logical blocks of 256 bytes each one. It is composed of two parts:
A.- file description
B.- data section

A.- File description
It occupies 37 blocks and it has the following structure:

- Allocation Directory	:	4 blocks. It describes the data blocks occupation for up to 8192 data blocks.
- Job Directory	:	1 block. It describes the Jobs, for up to 32 contemporary Jobs.
- Batch Directory	:	32 blocks. It describes the batches allocated to each Job, up to 64 batches per Job. Batch directory of one job is 1 block long.

B.- Data Section
It occupies up to 8192 blocks. A data record is at most 248 byte long. Each block has the following structure:

SDEF.1

- Heading : 3 words
- Data Record.

Fig. 1 shows the complete structure of the file

USER INTERFACE

The task SDEF maintains the file SDEF. Each operation request is made by a MACRO-II user program by means of a proper macro and by a FORTRAN user program by means of a SUBROUTINE CALL.
The operations provided are the following ones:

- Insert JOB in the 'JOB DIRECTORY'
- Delete JOB in the 'JOB DIRECTORY'
- Delete BATCH in the 'BATCH DIRECTORY'
- Open BATCH in ADD or CHECK mode
- Delete DATA RECORD
- Read DATA RECORD
- Write DATA RECORD in ADD or UPDATE mode
- Insert a new DATA RECORD
- Read DATA RECORD preceding the last one
- 'VIRTUAL' write DATA RECORD
- Close BATCH
- Close the 'SDEF-DAT' file

DOCUMENTATION

Product Specification - SDEF - INPRODE - 1977 SY0321

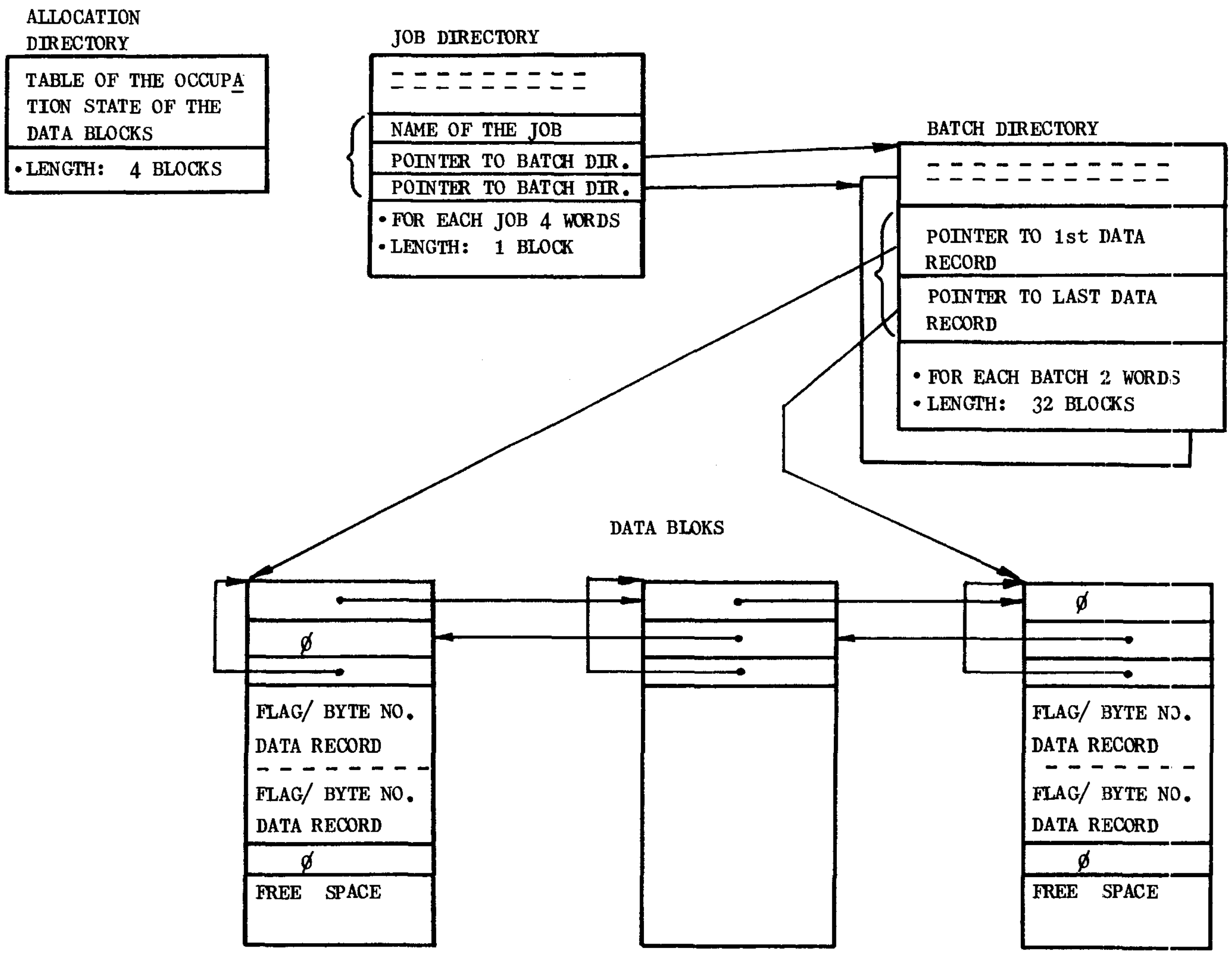

Fig. 1

SDEF 2.