PAPER SUBMITTED TO THE 7TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING.

SOME FRAMEWORK IDEAS FOR SOFTWARE ENGINEERING EDUCATION

BY F. SÁEZ VACAS

PROFESSOR ON COMPUTER SCIENCE AND CYBERNETICS

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN.

CIUDAD UNIVERSITARIA

MADRID-3

(SPAIN)

1. THE AXIOM OF INSTABILITY OF THE UNIVERSE OF DISCOURSE

It is difficult, if not impossible, to find something that is not changing in computer technology: circuits, architectures, languages, methods, fields of application... The "central object" itself of this brand of engineering, software, represents such a diverse reality (many objects) that the fact that it has only one name gives rise to considerable confusion. This issue, among others, was taken up by Fox (1) and, at this point, I would like to underline that it is more of a pragmatic issue than an academic one.

Thus, Software Engineering Education moves in an unstable, undefined world. This axiom governs and limits the validity of all educational proposals in the area of Software Engineering and, therefore, all the ideas presented in this paper.

2. A 3-P APPROACH IN SOFTWARE ENGINEERING

To start with, Software Engineering moves inside of a triology of categories: the problem, the process and the product (software). The inseparability of these three categories is a basic concept, a framework idea, and can be graphically presented by employing Morin's denotative system (2) (see Figure 1). As you can see, this inseparability is directed and dynamic.

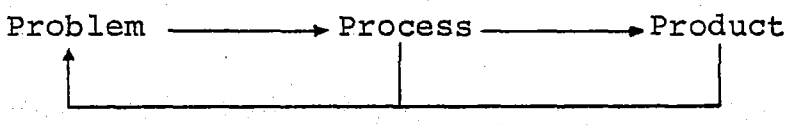

Figure 1. 3-p Diagram

In other words, the isolated handling of each of these categories is erroneous. The degree of error would be a function of the specific circumstance of the ontogenetic circuit, which is present in the diagram shown below. Although it may sound complicated, this is how it is.

As is the case with all education, Software Engineering Education operates in practice with simplifications, but nothing would justify its concealment of fundamental relationships. The diagram in Fig. 2 is a simplified enlargement of the diagram in Fig. 1. In this enlarged version we find the essential part of relationships between a problem posed, the software that automates its solution and the process that leads to the development of this solution.

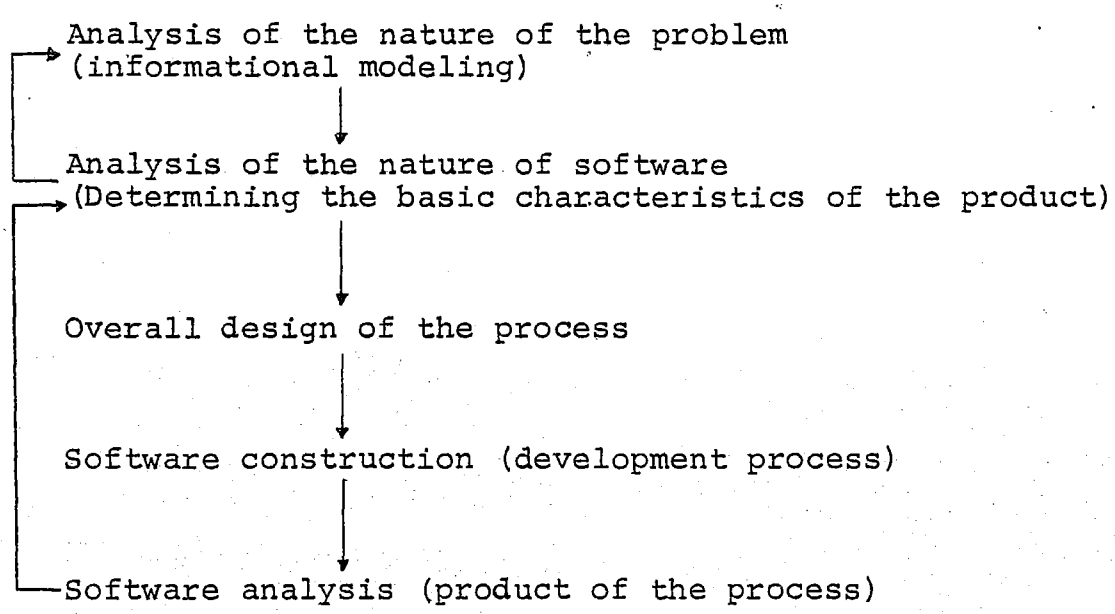

Figure 2. Simplified enlargement of the diagram in Figure 1.

All complex problems are solved by an ordered set of mental processes. In Software Engineering, the set must also be cost effective. Here we will call this set a 'process'. The diagram in Figure 1 is developed on a recurrent top down basis, this is to say that it is valid and always the same for each one of the temporal phases, which are defined by substituting

'problem' and 'product' for the input and output of each sub-process.

At the first resolution level, an overall design of the development process must be carried out, at the end of which we will obtain a product called software. We know that this product will display many characteristics that are dependent on the criteria that guided its development process. An old and simple experiment (Weinberg, 1974) (3) continues to be illustrative of the concept we wish to express. (See Figure 3) In general terms, it is important to previously mark off some parameters for the final product in order to give shape wherever possible to some general technical and organizational areas of this development process.

	Resulting Rank on Performance (1 = Best)				
Team objective: To optimize	Effort to complete	Number of Statements	Memory Required	Program clarity	Output clarity
Effort to complete	1	4	4	5	3
Number of statements	2-3	1	2	3	5
Memory required	5	2	1	4	4
Program clarity	4	3	3	2	2
Output clarity	2-3	5	5	1	1

Figure 3. Experiment by Weinberg-Schulman, 1974 cited in (3)

3. A FIRST APPROXIMATION AT DETERMINING THE MORPHOLOGY OF THE SOFTWARE ENGINEERING PROCESS.

In the previous section, we established, among other functions, that the process depends on the problem and the product. This shows that the techniques chosen in the process and the temporal distribution of the effort will adopt a thousand forms in practice.

Nevertheless, a standard concept, the software life cycle, and a number of associated estimation techniques, which appear to be independent of the problem and product, have become widespread. This is the way the concept is often times erroneously interpreted and applied.

In order to simplify its management, let us accept that it is desirable to divide the process into temporal phases. By paying attention to two "characteristics" of the problem -its expected rate of change and its relational complexity-, it is possible to get a first qualitative idea about the emphasis that will have to be given to specific dominant phases.

In Software Engineering, the difficulty involved in clearly establishing a system, a model or the general areas of a solution is an attribute of the problem (its complexity) and, in part, an attribute of the designer. The interrelationship of both attributes generates another essential characteristic of the problem, which I call the relational complexity of the problem. In my opinion, it is difficult to quantify this characteristic, but this difficulty does not make it less real.

This characteristic initially impregnates the process with a diverse degree of fuzziness, which fluctuates between merely repetitive and routine activities (minimal fuzziness) and the most profound and creative intellectual activity (maximum fuzziness).

Moreover, problems pose a greater or lesser demand (capacity) for change in their solution (product) over time. Although we can express it this way, we are well aware that the cause behind the changes is not only the problem in and of itself, but all the conditions of this problem, the formulation of this problem, the formulation of the solution, the implementation of the solution (product), the human or artificial environment in which the product will be used, etc., (4), in other words, the problem and its circumnstances. I call

this an essential characteristic of the problem, and I think that we should establish its order of magnitude in the first loop of the diagram in Figure 2. Outside of this context, this is a subject that deserves to be worked on theoretically.

These two characteristics produce some very educational general ideas when they are put in contact with a theoric distribution of the software life cycle in three large phases: system definition, implementation and maintenance. This is what Lehman (4) called them, and this is what I independently called them at a seminar I spoke at in the same year (5).

The first characteristic basically affects the first of these large phases, and the second characteristic the last one. The process must be designed as a whole, and its morphology is determined *ab initio* by the degree of importance of these two characteristics. We are going to develop this next.

It has been fully demonstrated that the definition phase is capital with respect to the results of the overall process. It requires a greater effort in problem solving, analysis and decision-making, in general. It handles techniques and languages that have barely been formalized or that have a very narrow range of application. The difficulty of the tasks implied, more so than the resources involved, determine its temporal distribution.

This last aspect could be illustrated in a special albeit graphical way. In a project having a manpower curve that fits the Rayleigh software life cycle curve (6), ($y' = 2kate^{-at^2}$; $a = 1/2t_d^2$; k = total accumulative manpower utilized by the end of project; t_d = development time), the shape parameter governing time to peak, a, is related to the idea-generation rate, in other words, to what in this case I call relational complexity. The greater the relational complexity, the smaller the value of a (the longer the project) and, therefore, the definition phase

will gain importance. Reverse reasoning says that if complexity is minimal, the definition phase will become minimal or disappear.

The software maintenance phase, and even its own existence, depends on the need for change generated by the problem. Obviously, one of the aspects to underline is that, in spite of the unfortunate name of maintenance, the general meaning of this phase is that of adaptation and evolution, in order to adjust the product to changes in the problem or to improve efficiency. The life idea is here, and it can be easily transmitted through a representation using a basic cybernetic diagram.

This simple shape (Figure 4) contains basic aspects that distinguish the results from Approaches A and B. Approach B deals with the matter as a dynamic system, in which the problem is something that changes with time. And this shows the natural evolutionary tendency of software, for the process continually feeds on the discrepancies between the software solution and the real needs of the problem (the ⊗ represents a

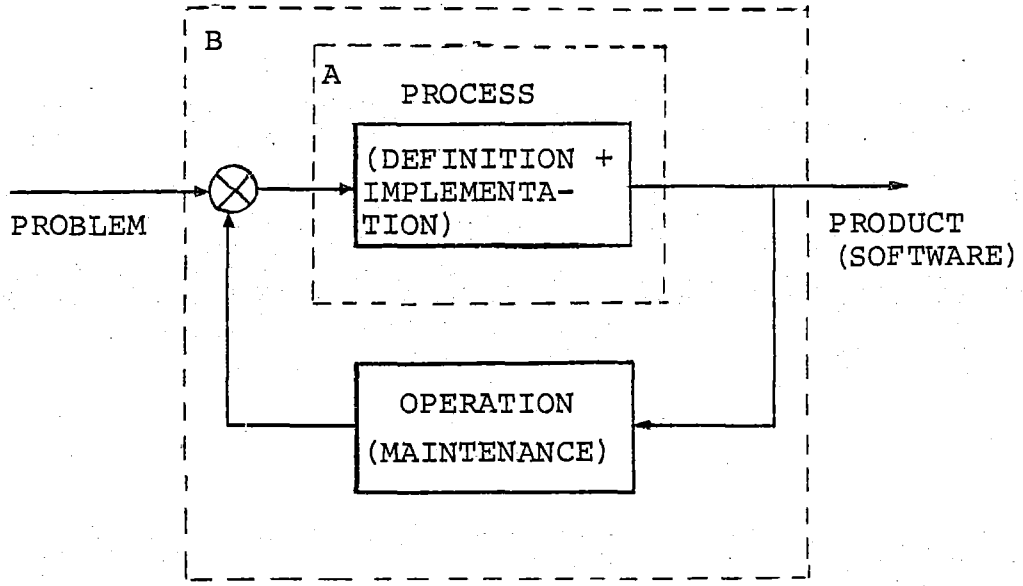

Figure 4. A cybernetic representation of the 3-p diagram.

comparator). Based on Approach B, the basic purpose of the process is to remove these discrepancies on a recurrent basis. In other words, the process is designed and optimized as a whole in order to achieve this aforementioned purpose. The 'process' block in Approach B in Figure 4 already "contains" the maintenance activity. Nevertheless, in practice Software Engineering presents a large number of cases in which software life cycles are addressed with the spirit and techniques of Approach A (7).

Using a very elemental, binary, logical (and we hope didactic) table, in Figure 5 we have summarized the dominant phases in the initial design of the process, according to the existence (=1) or non-existence (=0) of the characteristics of the 'rate of change' and the 'relational complexity' of the problem. (You should note that the life cycle only exists in the combinations of columns 2 and 4.

Phases of Process (Overall Design) \ Characteristics of the problem (rate of change, relational complexity)	00	10	01	11
FIRST PHASE	--	--	x	x
SECOND PHASE	x	x	x	x
THIRD PHASE	--	x	--	x

Figure 5. General triphase morphology of the process, according to the rate of change and the relational complexity of the problem.

Now that we have carried out this first approximation, enabling us to point out areas of attention, a study of the product's characteristics will be necessary to determine a

deeper technical analysis into the detailed planning of the process.

4. A 4-P APPROACH: THE DEGREE OF EVOLUTION OF SOFTWARE ENGINEERING, A LANGUAGE LEVEL ISSUE.

Sections 2 and 3 dealt with general ideas applicable to a software object, throughout its genesis and life. These ideas apply to ontogenetic processes.

Now if we think about the objects-software set, which over time has solved a specific kind of problem, we enter into an area of philogenetics. From a philogenetic point of view, the individual parameter which best measures the evolutionary degree of the engineering employed is (are) the language level(s).

We have characterized software engineering with a 3-p approach. The technological evolution of this 3-p diagram must inevitably refer to the power of the tools that people use to give material shape to the 3-p approach. These people include the analyst, the programmer, the project manager, the operator, the documentalist, etc. In order to carry out their respective jobs, each one of these people executes a series of mental opeations, whose complexity depends on the level of language defined for the job at hand. (This subject was generically evaluated by Halstead in (8) and other publications).

Programming languages, data definition and manipulation languages, specification languages, job control languages, software support tools, programming environments, hardware and software architectures... directly or indirectly all this is language.

Language is measured better in relation to man than in relation to machines. And the closer it is to man, as it concerns the fueling of the tasks implied in the 3-p diagram, the further evolutionary progress advances in the area of software tech-

nology and engineering. The "4-p = people-problem-process-product" diagram means that in order to solve a problem by means of a software object (product), a temporal process unfolds in which various people participate, coordinating the use of different languages as tools.

It is said that language is the house in which man lives. Language (artificial) is the house in which software lives.

We can assume that, in general terms, in Software Engineering the set of languages employed is a central theme. The real set of languages constitutes a skeleton on which, in each case, the four ps form the flesh.

5. 5-P APPROACH AND COMPLEXITY LEVELS

In section 3 we talked about the relational complexity of the problem, which is an aspect that was tied solely to the definition activity, in the first phase of the process.

In general, complexity may be a factor present throughout the entire Software Engineering diagram.

Algorithmic complexity and software complexity, among others, have been studied. Software complexity has received considerable attention due to the economic impact of software on the total cost of computer use. Among the experts who have addressed this subject, probably Halstead (8) and McCabe (9) are the best known. Sáez Vacas (10) has broaden this subject, by proposing a hierarchy with three levels of complexity.

In the first level, we would situate the software complexity which, in real terms, is applied to an isolated object -usually a program- of a set that we call software.

Above this element we find a group of interconnected elements. Examples: an operating system or a data base management system are program groups. The group is a system and it requires a systems approach. The emerging complexity is a systemic complexity. And, once all of this is fit into a 4-p diagram, the systemic complexity characteristic becomes more pronounced, due to the set of languages employed and the the set of people employing them. This is the second level of complexity.

In the third level, more complexity surfaces due to the possible involvement of another new set of people, the product users. The discordance between the people of the first set (who we will call the producers) and between the languages gives rise to a disruptive agent, disorder, which grows with the complexity of the system. Actually, disorder is another inseparable aspect of complexity, it is the disorganized face of complexity, which is prompted by unreliability, unresponsiveness, excessive costs and time periods, etc. Product users add wood to this fire when the relational complexity of product use is high, which is seen as misuse, insatisfaction, etc. (manifestations of disorder from the perspective of the technological world, based on logic and organization).

As you can see, the third level of complexity is applicable to anthropotechnical systems, where we can also clearly see disorganized complexity.

Finally, the diagram in Figure 1 should be developed into a 5-p diagram (Figure 6). We should also add that the set of possible users (the 5th p) is an inseparable part of the diagram. Moreover, we can assert that the study of complexity should be systematically and incisively introduced into high level Software Engineering Education.

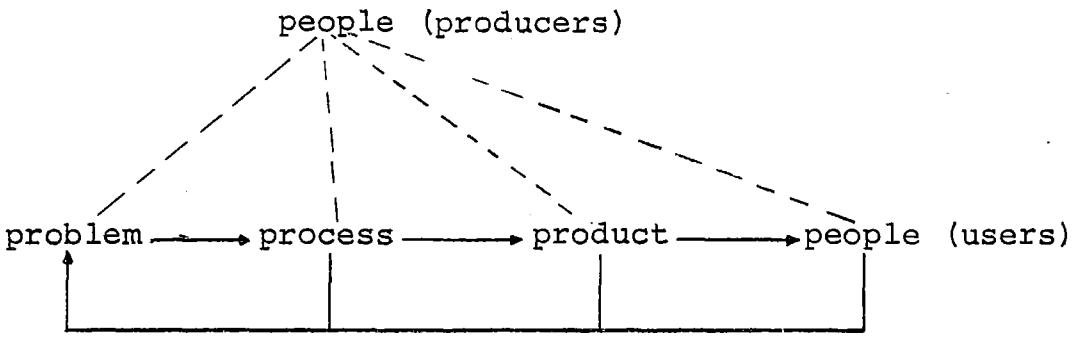

Figure 6. 5-p Diagram for Software Engineering.

REFERENCES

(1) J.M. Fox, Software and its Development, Prentice-Hall, N.J., 1982.

(2) E. Morin, La Méthode I: La Nature de la Nature, Seuil, Paris, 1977.

(3) B. Boehm, Software Engineering Economics, Prentice-Hall, N.J., 1981, pag. 20.

(4) M.M. Lehman, Programs, Life Cycles, and Laws of Software Evolution, Proceedings of the I.E.E.E., Vol. 68, No 9, Sept. 1980, pp. 1060-1076.

(5) F. Sáez Vacas, Seminar on Planificación y Control de Proyectos Informáticos, ERIA, Madrid, Apr. 1980.

(6) W. Myers, A Statistical Approach to Scheduling Software Development, Computer IEEE, Dec. 1978.

(7) F. Sáez Vacas, Factores Críticos en el Proceso de Desarrollo de Software y Expectativas Tecnológicas, Presentation at Conference COMPU-82, Quito, Oct. 1982.

(8) M.H. Halstead, Elements of Software Science, Elsevier North-Holland, N.Y., 1977.

(9) T. McCabe, A Complexity Measure, IEEE Trans. Software Eng., Vol. SE-2, Dec. 1976, pp. 308-320.

(10) F. Sáes Vacas, Facing Informatics Via Three Level Complexity Views, Communication accepted at 10th International Congress on Cybernetics, Namur, Aug. 1983.