Carlo Muzzì
Maurizio Caramellino

Configurative Programming

Abstract
The "Configurative Programming" is the proposal of a new paradigm to develop software based on the idea that applications can be realized without the formalisms and the tools typical of the traditional programming languages, in general domain of the professional developers, preferring an approach that foresees to configure a software environment potentially usable by whoever instead knows how to use a computer. This paradigm facilitates the activities of software development and maintenance because it favours the direct participation of the customers to the software production and simplifies the distribution and the use of the realized code.

www.configurativeprogramming.org

"Configurative Programming" is the English translation of the original Italian version entitled "Programmazione Configurativa".
Last significant revision August 2007. Digitally printed in Italy.

URL: http://www.configurativeprogramming.org
E-mail: authors@configurativeprogramming.org

Introduction

In the first section of this treatise we briefly deal with the evolution of the programming languages, underlining the meaningful improvements gotten at every single step of the evolutionary process and comparing the languages with the different methodologies proposed for the management of the software life cycle. On the basis of this analysis we expose, in the second section, the motivations that have conducted us to define this new programming paradigm.
In the third section we introduce the proposed axiomatic model as the theoretical base of this new paradigm, discussing the principles of it and analyzing the improvements introduced in the production of the software. In the fourth section we potentially treat of how such paradigm allows to define different programming languages, also allowing to extend the community of the developers, particularly we will explain how these languages will simplify and make faster the software development activities and the software maintenance activities.
The fifth section analyzes the first programming language defined with this new model, illustrating, with particular emphasis, the choices that have conducted to the definition of the most meaningful elements. We will conclude discussing some perspectives offered by the configurative programming.

I - The evolutionary process of the programming languages

The programming has been one of the principal areas of study of the computer science for a long time; in the last eighty years the different theoretical approaches used have conducted to the definitions of different paradigms and programming languages.

Initially, when still the famous computers did not exist the way they do today, scientific community was predominantly focused on the determination of the basic principles of Information Science. As in that period the philosophical aspects of the subject began to be framed inside mathematical models of reference, it was natural that also the approaches to the programming were also them primarily directed to mathematical models type.

The *Turing Machines*, proposed[1] by A.M. Turing, really constituted the typical example of a machine (or computer) meant to perform different algorithms but, typically, inside a predominantly mathematical field of application; in fact[i], Turing machine used some numerical systems to represent the treated information, the state assumed after every single activity it turns and the instructions that could be performed.
For the purpose of this treatise it is not necessary to deepen the basis of the theory of computation, instead it is remarkable to consider an important consequence of these ideal machines: Turing pointed out that it was possible to invent a more general single machine, called *universal computing machine*, able to simulate the activities of one any other Turing machine.

The idea that a calculator could be programmed thinking of working on another calculator introduced the concept of abstraction that allowed to avoid the use of the machine languages, also named *low-level languages*, replacing them with formal languages of higher level. It is known that the first calculators were programmed directly producing code in binary (the code was written with the so-called *first generation languages* or *1GL*) the following developments led to the use of symbolic languages or *assembly languages* (defined *second generation languages* or *2GL*), in which the binary strings were replaced by correspondents textual symbols: as examples of symbolic languages we can mention *IBM BAL* and *VAX Macro.*

The use of the symbolic languages induced however some substantial problems: the programs were always writings for the family of calculators that had to perform them, therefore they were dependent on a specific hardware; besides the programmer worked with tiresome mental operations that were different from those typically used by the human beings.

In the period between the years '50s and the years '60s experts exerted themselves trying to find solutions to such problems; these efforts were concretized in the definition of different programming languages, each mostly turned to specific application areas of the real world, that had in common a lexicon and a syntax nearer to the human level: for this reason defined as *high-level languages*[ii] .

These languages (known also as *3GL* or *third generation languages*) typically used key words and codes derived by the English language, in such way the source codes were more easily legible and comprehensible by a human programmer but naturally they could not be processed by a computer without a translation in its specific machine language that had to happen preventively or during the execution: initially the translation was realized in execution time through a component (called *interpreter*) who translated in the specific code machine; subsequently, the necessity of making the

---

i According to a conjecture of A. Church, note as Church's Thesis, the class of the problems that could be symbolized and solved with a Turing machine corresponded to the class of the computable functions; in effects the same Turing showed as the halting problem was not solvable: a Turing machine would be able not to answer if another machine will arrest him or less.

ii In the succession of this treatise we will see various classification of some programming languages; since also the languages as the other softwares, endure periodic revisions and updatings, it could happen that different versions of a same language belong to different families.

execution time faster and improving the abstraction process, led to the definition of the *compiler* concept: a dedicated instrument that it allowed a translation that came executed, in optimized way, one single time before the execution.

The high-level languages are also known as directed to the assignment (*task-oriented*) that is with reference to the environment in which the programmers would have had to operate; from this approach it derived a classification of the languages in three great families still used today:

- the *imperative languages* in which writing some code means to instruct the computer on "what to do" and "what to get"; the program is constituted by a sequence of instructions to be performed in pre-arranged order, through flow instructions, that they change the content of the memory of the computer (assignment of variables values, passage of parameters). In this class we find the more common historical languages, the ALGOL and the FORTAN dedicated to the scientific calculation and particularly used by mathematical, physical, astronomers; the COBOL for managerial type applications; the Pascal, the BASIC and the Modula-2 used in education field; the C for the operating systems and the games; the Ada used for military applications, critical mission, embedded systems, communication, CAD and finance;
- the *functional languages* used for the pure functional calculation; based on the use of functions that recall (recursion) or call other functions. They don't use the traditional concept of assignment of values to variables because the main data structure is the list. The best known language of this class is the LISP, particularly used in the artificial intelligence, in the military applications and in the world education, especially for its peculiar characteristic: that allows the direct implementation in a program of the computational model of the λ-calculation of Church[2];
- the *logical* said *declaratory languages* also because to the program is expected to demonstrate the truth of an assertion; the source code is constituted by a series of assertions of facts and rules without necessarily previously specifying the flow of execution, it will be the program to look for it according to the suitable objective. These languages don't have a specific field of application but they are particularly valid to solve problems that concern entity and relationships; the PROLOG is the most known declaratory language.

For the goals of our treatment it is important to consider the cultural hinterland in which the programming of the time was developed; in that period prevailing necessity was to have formalisms that allowed to easily share among different programmers what realized by a single component; it was therefore necessary to simplify the activity of maintenance of the code and to guarantee the retention of the know how: the requirement to render the development activity of the software less subjective lead to the definition of specific methodologies of work.

At the end of 60 years' the software development began to be considered nearly like industrial activity, the necessity to make to cooperate different programmers on the same project was implemented using typical methodologies of the industrial world of the time. Particularly the new development methodologies introduced were based on the concept of module; the process of software development started with the approach to decompose the initial problem in more parts or modules, each one developed independently from the other, that at last they were recomposed in a single final product. This model of software development was defined *waterfall model*[3] and, within the theory of the languages, it introduced the concepts of the date hiding, of the encapsulation and of the structured programming.

The objective of the structured programming was to simplify the writing of the code forcing the programmer to use few structures of control that guaranteed one entry point and one exit point; these object were reached through programming languages that, implementing the principles of the theorem of Böhm-Jacopini[4], avoided the deleterious use of the unconditioned jump (goto[5]) from which derived the harmful phenomenon of spaghetti code. ALGOL, Pascal, Modula-32, C, Ada they are some examples of structured languages.

A salient aspect of this new approach to the programming is the less importance given to the programming language used[6] as opposed to a greater attention towards the data structures and the algorithms used to computerize[7] a problem; the data and the algorithms could be considered in independent from the formalisms that every specific language used in order to implement them, therefore the programmers had to dedicate greater attention to them in order to obtain a better code.

The structured programming was therefore a theoretical model that had to lead to the realization of sturdy code; but the real experience demonstrated that the developed software did not always satisfy the expectations of the customers. Until the half of '80s the common opinion was that the cause of such inaccuracy derived from errors committed in one of the various phases[iii] of software development; therefore greater emphasis was given to the adopted methodology of development .

---

iii The typical approach previewed that the macro-activities of analysis, plan and development ulteriorly were subdivided in: study of feasibility, collection of requirement, analysis, planning, development, testing and put in exercise.

As the waterfall model used an approach in more phases in which one could pass to the following phase only after having concluded and validated the previous one, it was obvious that an error in the phase *i* caused errors in all the phases from the *i+1* onwards. It was decided therefore to use the *V model*[8] in which the macro-activity of analysis, plan and development they were faced from the general level to the particular level (in descending sense) and subsequently verified from the particular level to the general level (in ascending sense): in this way they hoped to reduce the errors because the activities near the customers, considered more subject to error, were verified before.

Likewise to the methodological processes also the programming languages had an evolution: the demand to use instructions with formalisms nearer to the human language led to the creation of the *fourth generation languages* or *4GL*: particularly useful in the scripting and the languages of interaction with the database.

Despite the corrective interventions on languages and methodologies, the real experience continued to demonstrate that the phenomenon of the software not in compliance with the users' expectations still existed; that convinced the scientific community to accept the idea that it was impossible to realize a code that could totally satisfy the expectations of the customers: for many it became clear that the process of software development was intrinsically an uncertain activity. To face this new awareness was thought to make to again evolve the methodologies, proposing new models of reference, each able to face the uncertainty depending on the specific necessities of the application domain that the software required.

The *incremental delivery,* with its evolutions[9], constituted a valid approach for those realities that wanted to receive as soon as possible a part of the expected solution: the software was realized and delivered to single pieces. Instead in the situations in which the technological uncertainty it cohabited with the uncertainty of requirement it was proposed the *prototyping model*[10] based on the preventive creation of a prototype of the final solution on which to deepen discussions and analysis.

The models proposed by the literature, from the waterfall type to the prototyping, also articulating in different way the phases of the development process used always a same sequential approach; however some application fields existed (multimediality, operating systems, simulation models of natural phenomenons, control of production processes, etc.) in which it was not possible to follow a sequential approach. For these cases the *spiral model*[11] was proposed: this model was created on the presupposition that all the remarkable elements of the project were carried on in parallel as long as they reached a specific moment (defined *milestone*) in which all the actors involved in the project had to analyze how much realized and make any correction; the development process was cyclical because after the first milestone all the activities had to restart again, always in parallel, up to the new point of milestone. This model was called "spiral" because to every cycle the error margin stretched to reduce itself until converging towards one optimal solution.

From the point of view of the theory of the languages these models brought to the definition of the paradigm of *object oriented programming*. The approach adopted by this paradigm[12] consisted in representing the reality with some entities (definied classes) containing both the data structures and the procedures (definied methods) that operate above; when a program created an element of a class it activated an object and different objects could communicate among them through messages. The basis of this theory derive from studies of Dahl and Nygaard on the language SIMULA[13] and they conducted to the diffusion of different object oriented languages such as Smalltalk, C++, SQLWindows.

The object oriented programming has been able to produce a substantial improvement in the activity of development of the code, but such improvement has happened with an increase of the level of complexity of the same code; since every single part of the problem had to be defined in precise way, it was necessary to manage languages with a lot of opportunities. To find enough programmers able to profitably use these languages constituted for a long time the greatest obstacle to the diffusion of this model. With the new millennium the complexity of the problems to be faced increased following the new requisite determined by the consolidation of the web paradigm: substantially the code was expected to be able to operate moving itself on a world net, that it had to integrate heterogenous environments and hardwares and that could not guarantee the speed and the typical reliability of a LAN.
To solve these problems a new class of languages was proposed for the software dynamic development, defined *dynamics languages*, to which some new methodologies of development notes were placed side by side, known as *agile methodologies.*

The introduction of Java constituted a typical example of language able to supply support to these demands. Necessity to produce software that could operate on heterogenous platforms hardware, was reached particularly emphasizing the principle of separation among the code and the hardware that performed it through the introduction of the concept of *virtual machine*: on every type of hardware a particular version of the virtual machine was obviously necessary but, if present, hardware various could execute the same code.
The used technique required that every class of a Java program was individually compiled producing a bytecode; in execution time, therefore to run-time, the virtual machine would have loaded the necessary *bytecodes* and, if required, it

would also have performed them on different computers: this technique resulted extremely profitable both in the multilevel architectures typical of the corporate environments and in the solutions that use the web[iv] as a model of reference.

More recently the objective of a further reduction of the level of complexity of the languages has conducted to the introduction of the concept of *Domain Specific Languages*[14] (*DSL*): for this new concept different definitions have been proposed, generally united by the idea that the DSLs are languages clearly dedicated to the resolution of particular problems of specific application domains.
The primary goal of the DSL aimed to reduce the existing distance between the problem and the code, that was achieved simplifying the structure of the language that stretches to model itself on a specific class of problems. In some circumstances the specialization of the language has allowed to directly submit its management to the experts of the application domain; for this reason these languages become known as languages for not-programmers or *end-user languages:* for example the macro language of the spreadsheets, the HTML, the syntactic specifications through BNF, the SQL.

The objective of a greater dynamism has also been pursued for the process of software development, that has been made simpler and light, introducing new methodological models defined *agile*[15] (*agile methodologies* or *lightweight methodologies*) whose purpose was to supply models of reference that constituted a reasonable compromise in the choice between code development without using methodologies of development (with the risk of the chaos and the absence of results) and code development using extremely rigid and heavy methodologies (with the risk of obtaining a different software from what the customers expected).

These methodologies[16] have accepted the fact that reality is in continuous evolution, therefore every problem must be faced with an adaptive approach and not a predictive one: it is necessary to adapt dynamically to the reality and not to try to plan what could happen. Besides agile methodology is directed towards the team of development and not towards the process; if the focus of the traditional methodologies aimed to define trials without taking into account of the development team that would have automatized them, with the agile methodology express it is demanded to support the activity of the development team. In particular the ability to develop progressively the code continuously, testing it and eventually rethinking about the choices made, closely is connected to the adaptive ability of the development team that will have to contain not only programmers and information technologies experts, but also the buyers, that is the customers of the demanded software.

Among the different methodologies agile proposals, the most important has been the *Extreme Programming*[17] (*XP*) of particular interest for the great emphasis it gives to the concept of the test. If the test activity has always been a phase previewed by all the processes of software development in the XP the test assumes a crucial role: the programmer will have to write the tests that will verify the code even before writing the same code. This approach, strongly directed to the test, is generally become known as *test-first.*

The attempt to simplify the activity of software development has also been the objective pursued by the *general purpose languages* or *GPL*, primarily through the adoption of visual or graphical tools. To the programmer these languages have supplied an integrated visual environment of development (IDE) that allowed to shape in rapid way the graphical user interfaces of the consumers (form, grids, text box, button, etc.) integrating them with the code written by the programmer. These languages have simplified the problem of managing object oriented classes hierarchies supplying some special visual tools; they allowed to develop code assembling and adapting some component softwares seed-worked purchasable on the market or available in the world open-source; they have facilitated the access to the documentation and the help supplying mechanisms that facilitate the creation and the use of available resources on the web. The following evolutions of Visual Basic, SQL Windows, Delphi, are examples of these languages.

The introduction of the *Microsoft. NET Framework* caused a wider diffusion of these solutions concerning; an additional component to the operating system that has allowed to have an integrated platform for the development of service oriented applications directed able to also operate in operational environments in which the web[18] plays a primary role. Of particular interest has been the introduction, also in this context, of the concept of the virtual machine already discussed for Java, implemented through the use of the *Common Language Runtime* (CLR).
The. NET Framework also allowed to abstract the language used for the development: a programmer has been able to already use a language known by him, select among those adherent to the specifications of the. NET Framework, to write code and to produce the version compiled for the CLR. C #, Visual Basic. NET, J #, ASP.NET, Delphi 8, DotLisps, are all examples of such languages.

iv The applet is a typical example of single part of code that a web server sends to the remote clients for a local execution.

Even if the term[v] is used in not uniform way, the languages of the last generation sometimes were known as *5GL* or *fifth generation languages.*

II - Our opinion around the opportunity of a new programming paradigm

In the preceding section we have shortly summarised the evolutionary footsteps made by the software; what described put in evidence as the development process has been constantly influenced by the need of *simplifying* the activity of realization and maintenance of the code.

This requirement has been pursued through the definition of several paradigms from which different development methodologies and multiple programming languages are derived. The mutual existing relationship between methodologies and languages is for a long time object of attention; particularly if we consider the methodology as the implementation of a specific model among those proposed for improving the processes of software development.
Even if there are cases in which the languages have anticipated a methodology, usually the times of evolution of the methodologies are quicker than the languages simply because they are studied by more people; in fact if relatively few individuals are interested in the programming languages (a subset of which much interested in the computer science), the methodologies that improve the development process are object of ampler searches because they derive from the application to the software of the more general theories of the management on the improvement of the production process.

This relation between process and language can be expressed by the measure of the complexity degree; the figure 1[19] puts the evolution of the complexity in relationship with the language and the methodology.

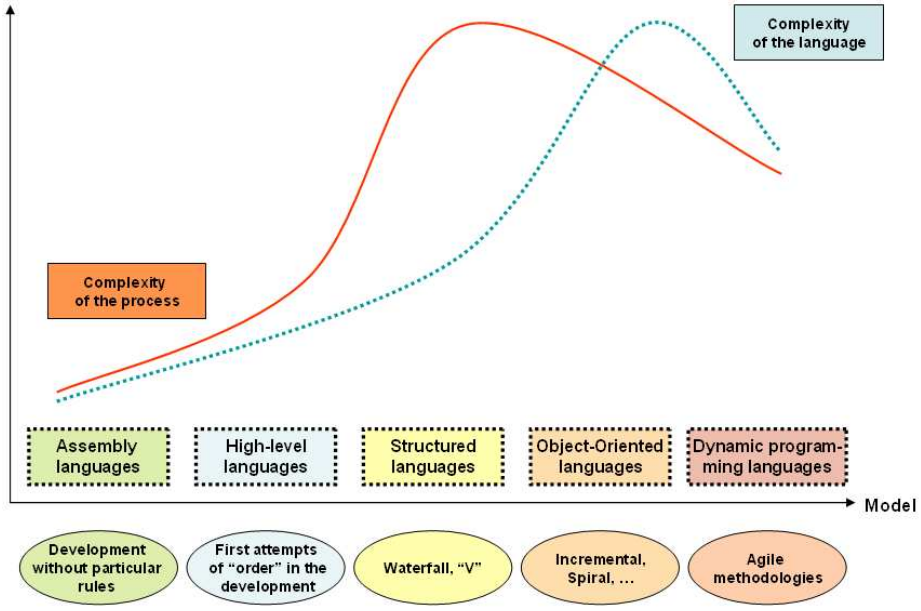

Fig. 1 – evolution of the complexity in relationship with the language and the methodology

The course of the graph underlines what already discussed; the simplification is a requirement commonly perceived that already it has been attempted to obtain; in effect the last models proposals have already lead to the definition of methodologies and languages that aim to reduce the problem of the complexity.

But we think that an further step towards the reduction of the complexity can happen only introducing new models of reference that allow to define methodologies and languages effectively simpler; but which can be the levers on which acting in order to complete this new evolutionary leap towards the increase of the simplicity?

We believe that an answer to this question can be found travelling over again the evolution process of the languages and the models discussed in the first part of this treatise; in particular dedicating greater attention to the languages rather than to the development methodologies. In fact experience induces us to affirm that a valid methodological support succeeds in exalting the effectiveness of a language but any methodology, however valid, can make the potentials of a specific programming language meaningfully evolve towards a superior level.

v Some authors consider these characteristics had already satisfied since 4GL intending the 5GL as the 4GL evolution towards the use of the bases of knowledge.

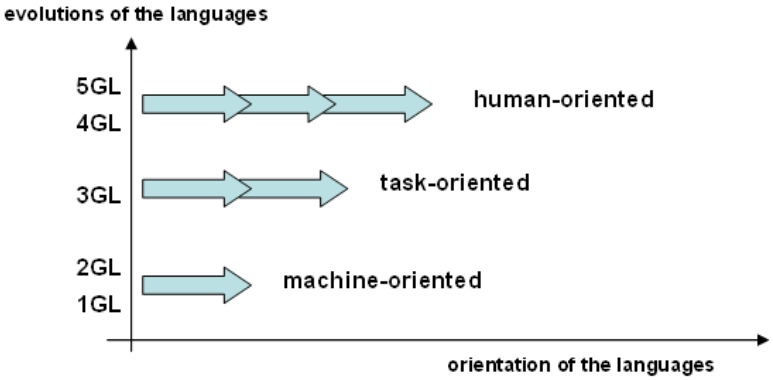

Fig. 2 – move of the focus of the languages during their evolutionary trial

Figure 2 puts in evidence, briefly, as the evolution of the languages has happened through the movement of their orientation; if the first languages were predominantly directed to the physical machine, the following generations have tried to move towards the requirements of the computers in order to approach more the human requirements. But if we analyze in detail what discussed previously on the so-called human-oriented languages, we find that these languages have been nearer the requirements of the professionals of the computer science rather than to the requirements of the generality of the human beings.

In effects, also sharing the objectives that some DSLs have tried to pursue in the attempt to approach the end users to the problems of the programming, we notice that this has happened to the detriment of an excessive reduction of the general potentials of the languages that have been confined in specific application domains: in fact the number of the potential users has been reduced inside to the experts of a single domain.

These considerations allow us to affirm that *the objective of the simplification of the language can be reached only defining some formalisms that allow to increase the number of individuals able to create new applications software.*

It is obvious that this objective is reached if we are able to answer to the following question: "Which characteristics must a programming language have to be defined simple?". To find an answer to this question it is again useful to examine the salient characteristics of the previous generations of programming languages.

In the first place we do not believe that the simplification can be reached using formalisms that supply evolved mechanisms of representation of the reality combined to a rich set of tools and commands to manipulate it: the history of the object-oriented languages has clearly demonstrated the validity of this approach, unfortunately it demands programmers with a remarkable experience.
On the other hand not even the use of formalisms logical abstracts allows to reach the objective: the same Turing machine, as others you approach[20] to the theory of the computability, they have demonstrated that a set extremely[vi] reduced of data structures and instructions could be diffused towards a vast group of individuals, but difficultly using them they could be realized the software usually present on the market.

Equally little fruitful, for our objectives, also it has been the experience to make the experts of the application domains participate to the definition of the languages. In 1959 the first version of the language COBOL[21] came really defined by a committee - "The Short Range Committee"- risen on initiative of the United States Government and a mixture of leaders of the primary public and private organisms of the epoch; the result has been a language of enormous success because it supports in an extremely valid way the financial demands, administrative and of the business (in fact COBOL is the acronym of "COmmon Business Oriented Language"), but whose use has always been domain of the programmers.

Therefore the last experience demonstrates as the attempt of the simplification of the languages can be only pursued with not traditional approaches; our opinion is that the *configurative programming* constitutes a new and meaningful approach to reach this objective.

Computer science uses the verb *to configure* in different circumstances, the *Cambridge Advanced Learner's Dictonary* defines it as "to adjust something or change the controls on a computer or other device so that it can be used in a particular way"; in our vision it allows to attribute an ampler meaning to the term "to program". We believe that the expression "configurative programming" can express a new model of the software development based on this idea: *it is possible to automatize an activity programming a calculator, in indirect way, through an execution environment (present on the same calculator) that is opportunely configured using a predefined set of elementary elements manipulated through rules that permit their correct use.*

---

vi In particular we refer to that ideal machine, defined Unlimited Register Machine (URM), derived from the studies of Stepherdson and Sturgis; this machine is extremely simple because it uses only four instructions and a single type of data constituted by an endless succession of registers.

In the next section we will define with more precision this new programming model using some axioms; as how this new approach can be used in order to extend the community of the developers it will be object of discussion of the fourth section of this treatise.

III – The Configurative Programming

The "*Configurative Programming*" is a new paradigm for the software development that doesn't use formalisms and tools typical of the traditional programming languages, but it adopts a new development model based on the principle of the configuration.

The axioms of the configurative programming are the followings:

1) we define "*Configured Application*" a whatever software realized through the model of the configurative programming

2) we define "*Execution Environment*" the infrastructure dedicated to the execution of a generic configured application

3) the execution environment has to allow the development, the distribution, the execution and the interaction of configured applications both on computer stand-alone and on multilevel architectures composed by calculators in various way connected in public or private nets

4) a configured application is developed through the allocation and the configuration of the elements supplied from the execution environment

5) the elements supplied by the execution environment can be composed, also repeatedly, to constitute more complex elements

6) the elements supplied by the execution environment can be configured in their own characteristics, in the actions that can develop and in the events which will be subject during their life cycles

7) we define "*Data Source*" the data structure (es. file, database, knowledge base) containing all the information that a configured application can manipulate

8) the set of the configurations that constitute a configured application is stored in a specific data source that we define "*Configurations Library*"

9) a configured application can access different data sources and it manipulates them through the elements supplied by the execution environment

10) the tools supplied by the execution environment to manipulate the data sources have to be independent from a particular implementation of the data source

11) the execution environment allows to manipulate the configuration library at the same manner in which manipulates the other data sources

12) the execution environment has to expose elements dedicated to the interaction between a configured application and another configured application and between a configured application and a specific software realized through a traditional programming language.

From the emphasis that the various axioms set on the concept of configuration it is derived the idea to define this new model of programming as "Configurative Programming"; we proceed, therefore, discussing about some remarkable concepts that derive from the analysis of the axioms.

We begin to underline that this model allows to develop the software through the configuration of the elements set to disposition of the execution environment; since it is the execution environment that allows the programmer to compose some instructions that automatize a certain activity, it is possible to assimilate the role of this model to that of a programming language. In reality the exact role of this model is ampler; to understand we observe the figure 3 recalling some concepts already discussed above.

We first notice that this execution environment has the ability to execute, at run-time, commands received from who programs it: therefore it is possible to consider it a particular type of interpreter.
The fact that the configuration happens on elements supplied by a software layer present on the calculator, a sort of middleware that beside the mechanisms of access to the data sources (database, file system, etc.) exposes also commands to manipulate them, allows us to think that it also the same aim as the CRL of Microsoft.

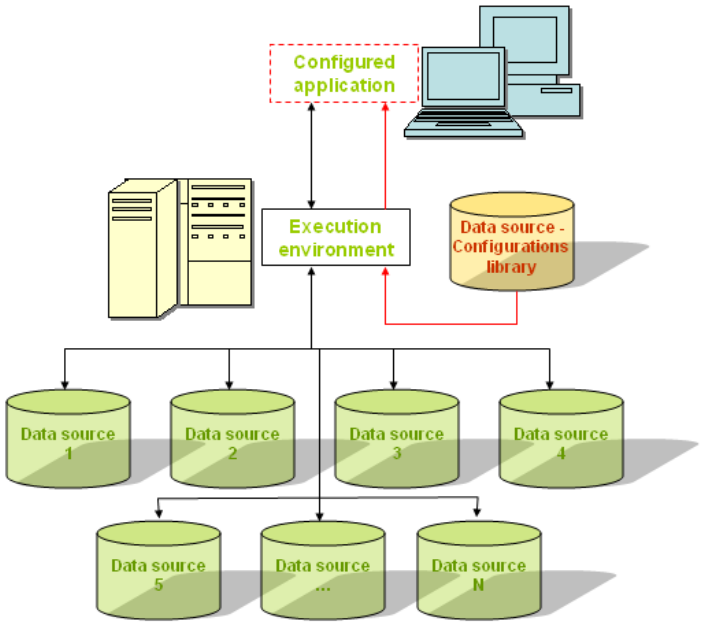

Fig. 3 – description of the configurative model

The execution environment can be seen, by his programmers, as an ideal machine with a similar aim as the virtual machine of Java; in fact it is possible that an application, realized according to this development model, runs on different families of calculators if every calculator has versions dedicated of the execution environment (versions that however must offer the same environment to configure to the programmers).

Therefore we can think how all these considerations are valid for a system in compliance with the principles of the configurative programming, for this reason it is correct to consider it as a *closed model* that includes:

1. the elements to be configured and the rules to manipulate them
2. the mechanisms that allow the software to be developed
3. the environment that performs the software.

Then, we can consider this system as a ideal machine able to perform some programs that are developed by the programmers and preserved in the configuration library. When the execution of a program is demanded, this machine will first load the necessary configuration from the configuration library, then it will use it to configure a new machine specifically dedicated to the execution of this program, finally it will execute the program activating the dedicated machine; in execution time the dedicated machine can interact with its users receiving input, elaborating data and producing output.
We immediately notice that this machine satisfies the requisite of the Von Neumann architecture[22]: therefore it is possible, on a certain point of view, to consider it as a concrete implementation of the universal Turing machine and to affirm that it has *computational power* analogous to the same universal Turing machine.

The typical characteristic of the Von Neumann architecture to treat data and programs in the same way, loading both in memory in execution time, finds a new application scenary inside of the configurative programming. While in a traditional programming language the software is written with formalisms and different tools by those commonly used for the data, in the configurative programming the software can be edited with the same mechanisms used for the data and preserved on the same mass storages (database, file systems, etc.). The code generation process is quickly in comparison with the DSL and is not required to embrace a specific application domain.

Substantially the expressive power of these languages, considered like ability to face wide classes of problems, depends on the implementation of the configurative principle adopted. We think that a configurative language is valid if it exposes the sufficiently wide mechanisms to cover the requirements standard of a single application domain or, alternatively, of more application domains that share a standard level of users; therefore, the correct definition of such mechanisms becomes the focal point to develope a configurative language.
As the technique of the constructions teaches us that starting from a little set of initial materials it is possible, using them in sufficent quantities , to build small or large buildings, equally we think that it is possible to define a little set of basic elements (the materials) and rules (the techniques of construction) to manipulate them that allow to face the challenge of the automation processes.
Besides, we believe that this facility in the code creation opens a new suggestive scenery in the software development: the model of configurative programming can become an extremely valid model to develop code in small pieces, for this it favours an approach both *adaptive* and *incremental.*

In fact, if a professional programmer is able to face and to develop an application through an approach to single footsteps, is also true that the consolidation of every footstep asks effort, knowledge and ability of vision that are not verifiable in the generalities of the computer users. Instead the configurative programming allows to a not professional programmer to really develop for single parts, because it allows to focus the attention on a specific aspect and to try immediately the effective operation without having to realize a single module software in a complete way; this

possibility actually derives from the mechanism of code realization that evolves from a process based on the writing to a process based on the configuration of the semi-finished elements[vii] supplied from the execution environment.

All that evidences as the configurative programming is particularly applicable with the most recent methodologies of software development: in fact the ability to sustain an adaptive approach makes it coherent to the principles of the agile methodologies. However this approach is also applicable with success with the most traditional methodologies, because:

- It meaningfully supports the development process based on the spiral development methodology; in how much it allows, after every milestone, to simply and easily modify big quantities of code modifying the configuration already performed.
- It is useful in the prototype model because it facilitates the creation of the prototype and moreover it allows to obtain various prototypes each usable in different experimental contexts.
- It satisfies the incremental model because the demand for a software for single pieces to supply to the customer sequentially is easily applicable both in the production phase of the single pieces and in the distribution phase (it will be enough to insert a new configuration in the configuration library).
- It favours the orderly participation to the writing process of the configurations of the various programmers that work on the same project, therefore it improves the results gotten by the application of the based methodologies on the models to V or waterfall.

The ability to successfully face the process of software development and maintenance, using different development models, allows us to affirm that the configurative programming is valid *independently* from the used methodology.

A further interesting reflection derives from the analysis of the last axiom: the requisite to have configurative languages that interact with other traditional languages is tied up to the opportunity not to consider the configurative model as a "panacea" for all the demands to computerize. The following section will show that the possibilities offered by the configurative model will be best manifested in the contexts in which the demands of vast communities of programmers will be satisfied; particularly we will see how by pushing this model to its extreme limit we would get some new languages whose use would be more complex than the traditional languages: the 12° axiom preserves us from such degeneration because it allows the integration of software modules realized through the configurative model with other software modules developed with traditional languages.

IV – Principle of the orientation of the configurative programming

In the preceding sections we have discussed and defined the model of the configurative programming, we will now examine as such model contributes to the increase of the community of developers; particularly it will be analyzed because this approach simplifies the process of software development and maintenance.

The analysis of the axiomatic model evidences as the base of this new approach to the programming derives having moved the focus of the programmer from a writing process to a configuration process; if such focus movement effectively simplifies the activity of software development or instead it makes it more complex depends on how much it succeeded to simplify the configuration process.
It is obvious that a extremely articulate configuration process, based on many elements everyone configurable in extremely meticulous way, will hardly conduct to the definition of a simple programming language, therefore it won't allow the diffusion of it on a vast community of programmers. As the demand to simplify the mechanisms of software development is connected with the demand to simplify the mechanisms to configure it is necessary to define some configuration levels that can are easily understandable and usable by different categories of users. This demand pushes us to affirm that the configurative programming can be considered as *oriented to individual* and therefore based on an approach *more subjective that objective.*

This consideration makes the assignment to define a specific configurative language extremely tiresome, simply because various individuals have a different idea of the "simplicity" concept; for instance: a mathematician could consider easy the use of numerical models, a philosopher could privilege logical models, a manager would probably appreciate chart models. On the point of view of the programming languages these different visions would be translated in the creation of different development environments; in our example the mathematician would probably choose functional languages, the philosopher towards logical languages, the manager towards macro languages dedicated to the manipulations of data charts.

---

vii Although these mechanisms are already present in precedents development paradigms (objects oriented, using component, etc.), the configurative programming extends to face them from different points of view.

Therefore we can assert that the implementation of the configurative programming paradigm can take place in different forms: each directed towards different categories of users. In this sense we can define that *the configurative programming constitutes an "Individual Oriented Programming."*

In order to define a language dedicated to a specific application domain it is sufficient to define a dedicate configurative programming language that is in a position to satisfy the requirements of a generic individual: an ideal representative of this category of users. It is obvious that the success of a such configurative language primarily will depend on the correct definition of this ideal individual, particularly having clear in mind its demands and operational ability, secondarily, having defined configuration mechanisms that a such individual will consider effectively simple.

Clearly, the attainment of this objective is the result of an empiric process based on the experience of those who will participate to the process of languages definition; the fact that human activities are manifold and therefore the application contexts can be manifold too, each composed by a plurality of single individuals, allows to infer that the processes of definition eventually undertaken could lead to the production of a big quantity of different languages that could degenerate in a new linguistic confusion.

The attempt to avoid one such degeneration as well as the demand to define languages that, likewise to the general purpose language, can be used in heterogeneous application domains, therefore to meet the demands and the favour of a vast community of users, induces us to propose the following definition: *the "User Oriented Programming" is constituted by all the implementations of the configurative programming paradigm that has the objective to allow to a generic end user to directly develop an own configured application.*

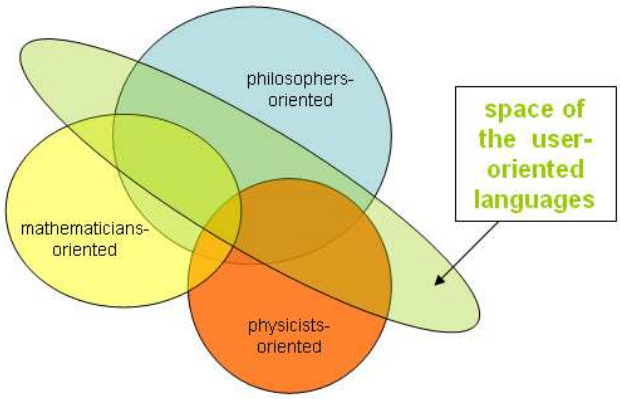

Fig. 4 – an ideal classification of languages oriented to different categories of individuals

Naturally also this definition is extremely ample and contains aspects of subjectivity; for such reasons we will now discuss of some general principles of reference that can lead to the correct definition of a programming language directed to the users.

A configurative language must expose mechanisms to be configured; in order to define the mechanisms that a programmer will be able to configure it is useful to establish the margins of the problem that is wanted to be automated. As already discussed above, the 12° axiom guarantees that a configured application can operate together with other applications developed with traditional languages. This possibility allows us to avoid to consider the configurative model as a panacea for the automatization of all the possible activities; it makes sense to use the configurative model only in the cases in which it produces of the effective benefits, basically in the cases in which:

- The participation of more individuals, with heterogenous professional competences, to the activity of creation and maintenance of software application; for example, in the realization of control systems of management for the companies.
- In the situations in which the software distribution model realized in configured way constitutes a strategic advantage for the producer of the software. For example, a software house could consider advantageous to realize with the configured model only the application infrastructure management (user's profiles, MDI, menu, command bars, simple forms of date-entry, query on the data, form to change password, support to reporting, operation in public and private nets, etc.) and to realize with traditional languages specific modules that contain a complex know-how or a know-how for which it is necessary a greater protection of the intellectual property.
- When the developers do not have the experiences and competences required by the traditional programming languages; for instance, in the development of applications for school or home use.
- In general in the cases in which the use of the configurative model allows to develop more simply complete applications or meaningful parts of theme.

In the other cases the use of the configurative model has to be valued in specific way to avoid that its application becomes self-defeating; for instance in the cases in which the management of a configured application becomes more complex in comparison to an analogous application developed with traditional programming languages.
These reflections remark that the set of computer projects realized with the configurative programming are extremely wide. Among the many possible fields of application we believe that a meaningful interest will be in the production of software for personal use.

As a consequence of the general diffusion of the computer technologies the houses of hundreds of million of individuals contain computational ability comparable to professional contexts; the last years have seen an exponential increase of the solutions software express realized and commercialized for the home consumer; instead, the possibilities that a single consumer can construct itself own applications that satisfy its personal requirements did not increase meaningfully. We believe that the configurative model will allow us to reach such objective.

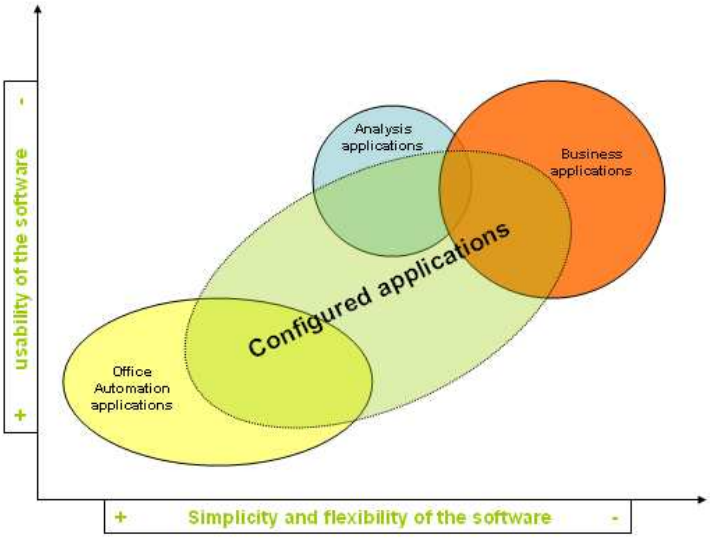

Fig. 5 – possible use of the model configuraivo in the organizations

Besides we believe that this model will also have a meaningful diffusion in the fields of the organizations that already have information systems. The figure 4 evidences the existence of a considerable space of diffusion constituted by those demands that traditional application solutions manage with notable economic and temporal investments, while the use of typical tools of the office automations (for instance, the spreadsheets) don't guarantee enough certainty in the data management (the information is destructured).

We believe that the two examples discussed, already expose with clarity the possibilities offered by this model. In general we think that the configurative approach is extremely valid in all the realities where it is necessary to manage in rapid and structural way the problem of the software uncertainty.

In the first part of the treatise we have discussed about the approach held from the different methodological models in facing the problem of the software uncertainty; the studies of Rittel and Webber[23] have shown as the reality is often constituted by *malignant problems* or *wicked problems.*

In effects, the real problems are observed from multiple points of view and with conflict levels of knowledge, therefore the software development, even if supported by valid methodologies, will hardly be totally in compliance with the requirements of the customers.

In this sense the adoption of the programming configurative, thanks to the intrinsic possibility to realize the software through an adaptive approach, allows to rapidly correct the imperfections found: for this reason it can be considered a winning approach in all those applications for which a shared definition of the problems[viii] between the actors who have participated to the phase of analysis has not been found.

For these reasons we can assert that it is possible to consider the configurative approach as a valid solution to solve the problems that, from the watefall[24] model onwards, often has compromised the processes of software development and maintenance.

V – The first language founded on the configurative paradigm

In the preceding sections we have introduced and discussed the configurative programming paradigm, underlining as such theoretical model innovates the activity of the code development. At the end of this discussion we think it is useful to shortly introduce the first configured programming language.

This experimental language, belonging to the class of the user oriented programming languages, has been realized by the authors of this treatise to accompany the proposal of the new programming paradigm together with a concrete implementation that shows the applicability in the real world of the proposed theoretical assertions. As it is not the objective of this treatise to deepen the analysis of solutions software we will confine ourselves to underlining some meaningful aspects.

We have seen that the objective of the user oriented programming is to define programming mechanisms that a generic user finds simple. Such objective has primarily been pursued using visual interfaces for the interaction with the users and secondarily basing the process of formulation of the configurations filling up of the tables.

The choice of the visual interfaces has been motivated with the obvious ascertainment that these have constituted the lever that has allowed the diffusion of the computers towards the generality of the human beings. Instead, the tabular representation has been adopted for the ability to conjugate the simplicity of use to the possibility to analytically treat a big quantity of data. In substance the basic idea of the spreadsheet has been reconsidered and enriched of further elements to support a more structured information management. As an example, the not professional programmer will

viii The known and encoded problems, that is somehow governed by the applications, they are also known as *tame problems.*

be able to try ideas and suggestions in other applications or documentary sources and to insert them in the own applications simply using the techniques of the "copy and paste": undoubtedly a possibility to develop code which appears very simple and suggestive.

From the requirement to use a rich graphic environment, already used with simplicity by most individuals, it is derived the choice to use an interface Windows Forms[25] rather than a browser (the use of the browser would also have required further competences for the management of the support infrastructures).

Further element of interest is the technique chosen for the interaction with the sources given: in order to favour the operation between computers interconnected on public nets at a low speed the logic of data management in disconnected[26] mode has been preferred (even if more evolved users can use more traditional mechanisms for the interaction with the DBs).

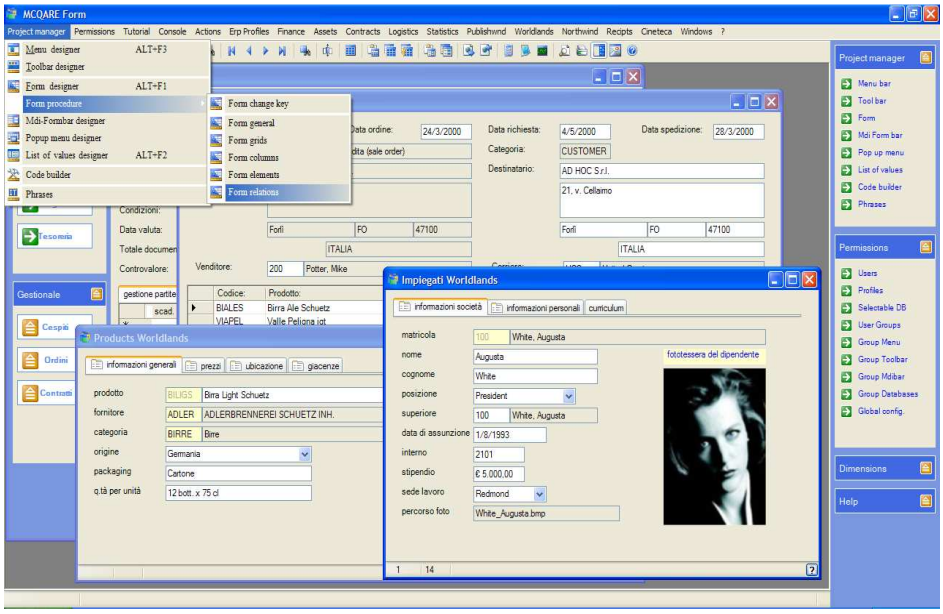

Fig. 6 – an example of forms realized with this new language

The demand to allow the interaction with other languages has been satisfied enriching the environment of an inside programming language of traditional type; an imperative language expressly defined for not professional programmers: with little effort, they would have at least be in a position to construct software small and simple (naturally, the professional programmers will be able to develop code more complex).

The figure 6 is an example of forms software realized through the language shortly examined in this section; we believe that this image, alone, witnesses the validity of the specific language, but above all you underline the possibilities offered by the new model of configurative programming to the professional programmers and in general to whoever desires to be able to autonomously develop an own application working software in an application domain of interest.

VI – Conclusions

In conclusion of this treatise we belive a positive conclusive judgment can be expressed on the validity of the configurative programming paradigm. We believe that such paradigm, in its entirety, constitutes a new way to the software development: it will allow to extend the competence of the code development towards wider communities of individuals, while the traditional actors of the software development will receive notable advantages because they won't have to deal themselves with all the connected aspects of the process of software development and maintenance but they can dedicate their own specialist competences to the more strategic aspects.

It is obvious that the ability to introduce simplification is directly proportional to the ability to implement the model producing some languages with a reasonable level of complexity. To define these languages it is necessary not to start from the presupposition to use them in all the circumstances, even with the collateral objective to expel the professionals of the software development from the automatization processes: who tried to define such a language, also in the cases in which he succeeded in the enterprise, probably he would obtain a new general purpose language: a language probably more complex than a traditional programming language.
Instead, it is reasonable to expect a simplification in the management of these superstructures[ix] that we also find in software created for modest requirements of automation: requirements that usually create real applications more complex than those demanded by the specific problem to automate.

---

ix Example of these demands, particularly present in the corporate environments, it is the compliance to the various national and international legislations (Systems of Quality, Data Protection, Computer Security, etc.) and the requirement to operate in Internet; business requirements that often demand complex infrastructures of support integrated with flexible mechanisms to manage the user's profiles, roles and software.

However, generally speaking, whoever uses software realized with the paradigm of the configurative programming will get benefits even in case an active participation to the phases of code development and maintenance is not desired. Primarily the possibility to facilitate the code maintenance allows to hypothesize remarkable economic advantages: if the laws of Leham and Belady[27] remember us that every software, to be able to satisfy its own users in long run, must be submitted to a periodic activity of updating, several studies[28] have clearly shown how the software maintenance absorbs a conspicuous part of the general costs sustained for them, even recent[29] analyses have underlined as this percentage you sometimes overcome the 90% threshold.
To the meaningful economic advantages the benefits of strategic nature must be added, derived by a greater *independence* of the software realized with the configurative model:

- The possibility to memorize the edited code equally in which the data are memorized, make it easier to preserve and transfer the know how contained in every single software. Besides, the technical possibility to consult and to modify in every moment the source code will allow the customer to contract more extensive licence agreements, that can reach to guarantee the real maintenance of the intellectual ownership of the code developed on request inside the organization of the customer, avoiding excessive dependences from single programmers or software house.
- It will simplify the evolution of the applications in the long run: a new version of the environment of execution, for instance necessary for an updating of operating system, can be installed without necessarily modifying the programs developed that they will stay unchanged in the configuration library; at the same time a change of the configured applications, to replace obsolete commands with others more advanced, can be carried out for instance with simple commands of updating of the configuration library.
- An interesting benefit will be obtained in all those contexts that need a lot of different applications but potentially integrated (ERP, MRP, CRM, Human Resource, etc.); the possibility to construct these applications using a same configuration environment, constituted by the same basic elements, will allow to realize software systems normalized and standardized: then easiest usable by different users.
- The configurative programming also allows to face entirely the matter[x] of the *legacy dilemma*[30] with a new approach: the code maintenance in the long run can happen in less critical way both for the facility to access the content of the same code, by itself originally organized in an orderly and documented way, and for the possibility to adjust, or also to rewrite, the existing configured applications simply by modifying parts of the configurations already realized.

Naturally the configurative programming does not only involve benefits. We have already seen that it does not have to be considered like an applicable panacea in all the circumstances; in some circumstances its use could result superfluous or even harmful. Particularly we believe that the use of the configurative model has to be valued with attention in the softwares in which the algorithmic element prevails in comparison to the simple interaction with the data: for instance, in the software containing *business rules* extremely articulated and dedicated to single domains. In these cases, the use of traditional languages, even limited to the development of specific modules to be inserted in an ampler system realized with the configurative model, could result more convenient.

Instead, it is generally probable that the configurative programming will favour the diffusion of new products, new techniques and new methodologies thanks to the real possibility of participation of the final customer to the process of software development and maintenance.

---

x The organizations that have the requirement to continue to maintain working in the time the own systems legacy must face the following dilemma: to restructure completely the systems legacy to continue to use them or to rewrite them trying to reproduce the same functionalities.

References

1 A.M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, in "Proceeding of the London Mathematical Society", XLII (1936). Light also *A correction*, ibidem, XLIII, 1937.

2 A. Church, *The Calculi of Lambda-Conversion*, in "Annals of Mathematics Studies", n°6, Princeton, 1941.

3 W.W. Royce, *Managing the Development of Large Software Sytems: Concepts and Techiniques*, in "Proceeding of the WESCON", 1970.

4 C. Böhm e G. Jacopini, *Flow Diagrams, Turing Machines, and Language with Only Two Formation Rules*, in "Communications of the ACM", Vol. 9, n°5, 1966.

5 E.W. Dijkstra, *Go To Statement Considered Harmful*, in "Communications of the ACM", Vol. 11, n°3, 1968.

6 N. Wirth, *Sistematisches Programmieren*, Teubner Verlag, Stuttgart, 1972.

7 N. Wirth, *Algorithms + data structures = programs*, Prentice-Hall, 1976.

8 P.E. Rook, *Controlling software projects*, in "IEEE Software Engireering Journal", n°1, 1986.

9 T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

10 J.E. Urban, *Software Prototyping and Requirements Engineering*, Rome Laboratory – DACS, 1992.

11 B. W. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, Vol. 21, n°5, 1988.

12 B. Stroustrup, *What is "Object-Oriented Programming"? (1991 revised edition)*, AT&T Bell Laboratories, 1991.

13 O.-J. Dahl e K. Nygaard, *SIMULA- a language for programming and description of discrete event systems, introduction and user's manual*, Norvegian Computing Center, 1965.

14 L. Walton, *Domain-specific design languages*, 1996. URL: http://www.cse.ogi.edu/~walton/dsdls.html

15 Various Authors, *Manifesto for Agile Software Development*, 2001. URL: http://www.agilemanisfesto.org

16 M. Flower, *The New Methodology*, 2000. URL: http://www.martinfowler.com/articles/newMethodology.html

17 K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

18 J. Richter, *Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web*, MSDN Magazine The Microsoft Journal for Developer, Vol. 15, n°9, 2000.

19 G. Succi, *L'evoluzione dei linguaggi di programmazione: analisi e prospettive*, in "Mondo Digitale", n°4, 2003.

20 N.J. Cutland, *Computability: an introduction to recursive function theory*, Cambridge University Press, 1980.

21 U.S. Department of Defense, *COBOL, Initial Specifications for a Common Business Oriented Language*, Government Printing Office, 1960.

22 J. Von Neumann, *First Draft of a Report on the EDVAC*, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

23 H. Rittel e M. Webber, *Dilemmas in a general theory of planning*, in "Policy Sciences", n°4, Elsevier Scientific Publishing Company, 1973.

24 P. DeGrace e L. Hulet Stahl, *Wicked Problems, Righteous Solutions: A Catalog of Modern Engineering Paradigms*, Prentice-Hall, 1990.

25 J. Prosise, *Windows Forms: A Modern-Day Programming Model for Writing GUI Applications*, MSDN Magazine The Microsoft Journal for Developer, Vol. 16, n°2, 2001.

26 D. Sceppa, *Microsoft ADO.NET (Core Reference)*, Microsoft Press, 2002.

27 M. Lehman e L. Belady, *Program Evolution: Process of Software Change*, Academic Press, 1985.

28 J. Koskinen, *Software Maintenance Costs*, 2003. URL: http://www.cs.jyu.fi/~koskinen/smcosts.htm

29 R.C. Seacord, D. Plakosh e G.A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, Addison-Wesley, 2003.

30 K. Bennett, *Legacy Systems: Coping with Success*, IEEE Software, Vol. 12, n°1, 1995.