Acta Cybernetica **16** (2003) 293-314.

Programming by steps

Raluca Oana Scarlatescu*

Abstract

The paper introduces a new method of software analysis, design and programming based on a different implementation of a logical flow: the sequence of steps is memorised in a database table, and each step is linked to a specific function inserted in a library. A main application manages the steps' information and runs the functions, until the steps are finished. The database management system stores the data of the each step and its precedence rules, the functions and their parameters, the static and dynamic values of the parameters, the errors, etc.

The paper details the principles of the "Programming by steps", explains the reasons, which originally motivated the development of the method, and defines the principal requisites to build an application system. Future aspects of the implementation, as well as advantages/disadvantages of design, implementing and maintaining the system are stated.

The paper includes a comparative analysis between the "Programming by steps" and another two methods of software engineering: the "Rapid Prototyping" and the "Component-based Design". Integrative comments and conclusive remarks are provided in the conclusion of the paper.

1 New changes in the evolution of the software methods for analysis, design and programming

The fast evolution of technologies reflects a boom of software applications' requests. The result is a larger application domain, which is more diversified and/or specialised, that requires new software methods for analysis, design and programming in order to develop open applications, which may be transparent and easy-exportable from one domain to another.

Other characteristics of the actual requests are the speed, the on-line and real-time features, based on the Internet and telephony's services. In these cases, it is necessary to find out solutions to permit the adaptation and the improvement of the existing software without interrupting the system. Also, it is recommended to maintain the operative execution by trying to control and reduce the hardware costs.

*PhD Student of the Academy of Economic Studies, Bucharest, Romania, e-mail: oana.rs@tiscali.it

The growth of the number of the applications[1] and the increased complexity of the information systems may produce problems for the analysts and programmers. Therefore it is necessary to develop new methods that shall simplify the work for the programmers.

"Programming by steps" is a method that can be used in order to analyse and design software in the following cases:

- for systems characterised by many applications belonging to the same family with common functions, or that may become common after a standardisation process;
- for real-time systems, where changes or maintenance should not cause the interruption of the existing services;
- for real-time systems, where it is necessary to test the newer releases without interrupting or damaging the older ones.

"Programming by steps" offers some advantages:

- standardisation of the problems in the analysis process, reducing time and resource consuming;
- higher degree of flexibility and greater speed in the programming activities for the applications in the same domain;
- better visibility that helps the control and error correction activities.

The method is based on the old flow-charts in order to obtain, not only a logical representation of the problem to be solved, but also the real-time functioning of the software itself.

2 What is "Programming by steps"?

Supposing one should write several programs that utilise functions belonging to a defined group and which show some homogeneities.

The traditional solution is based on:

- grouping several functions into a library;
- projecting more specific applications which call for different single functions with defined parameters, each one to solve some problems.

This solution supposes that a programmer writes the code for these applications: one defines the new variables, chooses and picks out the necessary functions from the library, integrates them, and after tests the program's functionality (see Fig. 1).

[1] It is very difficult to imagine any activity that could not be usefully touched by the information technology, even if the effective informational process is not concluded yet.

In such a situation the difference between one application and the other is represented by the used functions' set and their priority. How could this work be done without writing a new program code every time? Is there an alternate solution?

The alternative is offered with the method "*Programming by steps*": a unique application accesses the functions' library through its own interface. The application consists of an "engine" that follows different logical flows memorised in either internal or external tables. One flow is used for every problem to be solved.

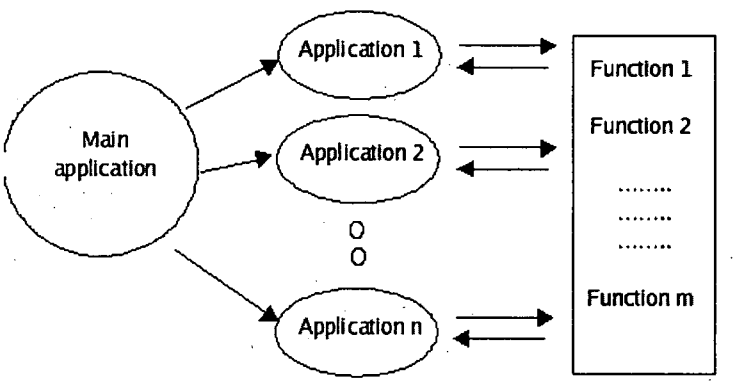


Figure 1: A main application launches many specific applications using the same functions' library

The analysis process of a new problem may be resumed in building up and implementing the logical flow in data tables. The execution "engine" pursues this logical flow and launches the various functions, building up the application by itself in a stepwise manner during run-time (see Fig. 2). Indeed it is necessary that the main application find all the functions inserted in the logical flow within the functions' library.

The design and implement phases supposed for every new application:

- to define the logical flow represented by steps to be done and their relative functions;
- to identify the values of all functions' parameters and the link/order among them;
- to put these elements together through their descriptive information inserted into a specific structure of tables (subsequently described);
- to launch the main application with a reference to the tables specific to certain applications.

The major advantage is that the main program is independent from the content of the logical flows. It knows only the modality to pursue such a flow, and to

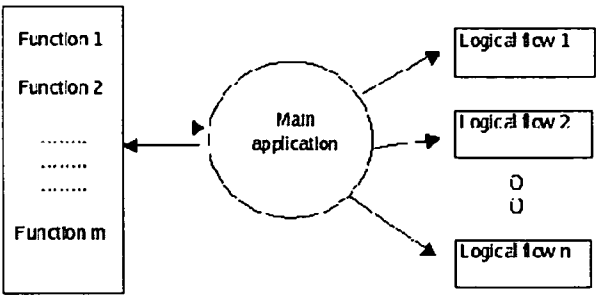


Figure 2: A main application is used for many specific logical flows, and it is linked to the functions' library for this goal

transfer different values from one step to another. Several concepts which are necessary to comprehensively understand the method "Programming by steps" are presented below.

2.1 Logical flows and flow-charts

In order to solve a specific problem it is necessary to understand its prerequisites and the different situations that could appear or influence it, and to intercept its possible results. It is seldomly possible to represent the solution through a *logical flow* of events or situations.

At the end of the 60's, Edsgar Dijkstra and others proposed three logical constructions to represent the logical flow of a program. These are: "sequence, condition and repetition. The *sequence* implements the procedures' steps, which are essential for each algorithm. The *condition* provides the capacity to elaborate selectively, based on determined logical conditions; whereas the *repetition* consents to perform cycles. All these constructions are fundamental for structured programming, which is an important technique of projecting the components' level. In practice, every construction has a predictable logical structure with the entry at the superior part and the exit at the inferior part, which allows to easily follow the procedure's flow"[2].

"Each programme can be projected, independently for the application area or technical complexity, by using only three types of structured constructions"[2]. Its *logical flow* provides a precise specification of the elaboration that means the events' sequence, the iterations, and the decisions' points, using certain data structures.

The *flow chart* is "a pictorial representation of the steps in a process, useful for investigating opportunities for improvement by gaining a detailed understanding of

[2]Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, pp.442-443

how the process actually works"[3].

"Flow charts have been used for so long that no one individual is specified as the *father of the flow chart*"[4]. "The flow chart is a means of communicating information. It must be able to communicate the steps in a process clearly and unambiguously"[5].

The New Oxford Dictionary defines the *flow-chart* (flow diagram) as: "a diagram of sequence of movements or actions of people or things involved in a complex system or activity; [...] a graphical representation of a computer program in relation to its sequence of functions (as distinct from the data it processes)"[6].

As mentioned above, for each initial application from Fig. 1 a set of information describing the own logical flow is presented: its steps and its functions, which have to be performed at every step.

2.2 Functions

The second part of the flow chart definition written in the New Oxford Dictionary introduces another important concept: *the function.* According to the same dictionary, the *function* is "a basic task of a computer, especially one that corresponds to a single instruction from the user"[7], or, mathematically, "a relation or expression involving one or many variables"[7]. Usually, a function is a "black box" that returns a value[8]. The user interacts with the function through the function's parameters[9], and possibly through global variables.

In "Programming by steps" a *function* is a routine (application, relationship, or transformation) that accepts a certain number of input parameters, uses them for its elaboration and returns some results through its output parameters.

Let us consider a function f_j, having x_j input parameters and y_j output parameters:

$$\begin{aligned} f_j : I_j -> O_j, \qquad & 1 \le j \le n \quad j,n \in N \\ I_j = I_1 x I_2 x ... x I_h x ... x I_{x_j} \qquad & 1 \le h \le x_j \quad h, x_j \in N \\ O_j = O_1 x O_2 x ... x O_l x ... x O_{y_j} \qquad & 1 \le l \le y_j \quad l, y_j \in N \\ f_j = f_j(i_1, i_2, ..., i_h, ..., i_{x_j}, o_1, o_2, ..., o_l, ..., o_{y_j}) & \end{aligned} \tag{1}$$

[3]See ISO 9004-4 "Quality management and quality system Elements - Part 4: Guidelines for quality improvement", 1993, p. 13. This standard has fixed a symbol set that ensures instant recognition by everyone in order to allow a unique representation of the fundamental constructions and logical flows.

[4]See J. R. Clauson, T. Glenn, J. A. O. Hunter , "Index of quality control tutorials", 1995, p. 2

[5]See T. Burns, "A Fresh Look at Flow Charting", p. 1

[6]See "The New Oxford Dictionary of English", Clarendon Press, Oxford, 1998, p. 707

[7]See "The New Oxford Dictionary of English", 1998, p. 743

[8]In programming a function may return a value, a vector or structure. The mathematical definition becomes larger. We know that a function may return a value or a memory's address (a pointer), that would contain everything. The large concept of functions is covered by functions and procedures in some programming languages.

[9]The parameter may have different values (see note 10): number, string, date, vector, matrix, pointer, structure, image, etc.

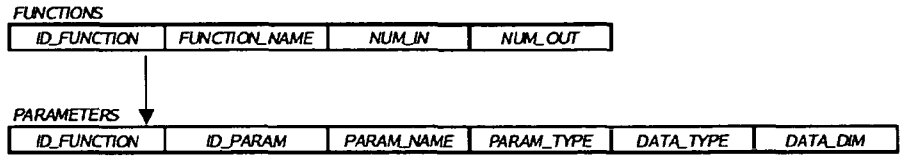


Figure 3: Functions and parameters

where
i_h – the input parameter, $i_h \in I_h$
o_l – the output parameter, $o_l \in O_l$
x_j – the number of input parameters for the function f_j
y_j – the number of output parameters for the function f_j
I_h, O_l – sets of values having homogeneous types of data (see the last footnotes)
Let us consider a set of functions defined above:

$$F = \{f_j\} \qquad 1 \leq j \leq n \quad j, n \in N \tag{2}$$

where
n – the number of functions in the library
Each parameter, i_h or o_l, is characterised by its type and dimension. In "Programming by steps" the parameters are like doors, which permit the entrance and exit of values in the function's body. Let us consider the generic parameter p_i:

$$p_i : PT * DT * N -> P_i \qquad 1 \leq i \leq x_j + y_j \quad i, x_j, y_j \in N \tag{3}$$

PT = {Input, Output}
DT = {Boolean, short, long, string, pointer, etc.}
$P_i = I_h$ or O_l
$p_i = p_i(t_i, dt_i, d_i(dt_i))$
where
t_i – the type of parameter p_i: input/output
dt_i – the data type contained by the parameter (Boolean, short, long, string, pointer, etc.)
$d_i(dt_i)$ – the dimension of the parameter (that may be dependent of the data type), $d_i(dt_i) \in N$
PT – the set of parameters' types
DT – the set of data types that depends on the programming language
One function may be a simple function or a macro-function that performs a certain group of activities. For example, it could only print a value on the output terminal or play a welcome message on an answering machine.
The functions are integrated in a library and called by a main application (see section 2.4). The information that is related to the functions' definition is represented in a database system in this modality:
The first table, FUNCTIONS, stores information about the functions:

- the function's identifier;
- the function's name;
- the number of input/output parameters (x_j, y_j).

The second one, PARAMETERS, contains information about the parameters of every function: the function's identifier, the parameter's identifier, its name[10] and type (input/output), the accepted types[11] and the maximum size of respective types[12].

2.3 Steps

Returning to the structured programming, the body of a program (between start and stop) is composed by a sequence of steps.

"Programming by steps" considers a *step* as the smallest logical part of a program, which can be associated with a function to be executed. The decomposition's level of a logical flow in its steps is inversely proportional to the standardisation's degree of the functions: the more abstract the functions, the fewer are the steps.

Consider S the set of steps from a logical flow:

$$S = \{s_i\} \qquad 1 \leq i \leq m \quad i, m \in N \tag{4}$$

where

s_i – the step

m – the number of steps to be done

Each step s_i is described by the following information:

$$s_i = s_i(t_i, \{(c_{ik}, s_{n_k})\}_{1 \leq k \leq n_i}, f_i) \qquad 1 \leq i, n_k \leq m \tag{5}$$
$$s_i \in S, \quad t_i \in T, \quad f_i \in F, \quad c_{ik} \in C, \quad s_{n_k} \in S \quad i, k, m, n_k, n_i \in N$$

where

t_i – the type of step s_i

f_i – the function associated with the step s_i

[10] A generic name is used in order to identify the parameter.

[11] The types and dimensions depend on the used programming language and/or database management system. In this example SQL Server 2000 and C are considered. The information is only descriptive, and it should be used during the data input process or the program's execution. We note that using a varchar data type, it is possible to memorise different data in the database, and after to utilise them independently of their types, because of the implicit conversions done by the SQL Server 2000.

[12] It is recommended to use a type compliant with a major number of possible received values and to perform internal conversions either at the function calling level (main application) or at the function execution level (library).

$(c_{ik},\ s_{n_k})$ – one of the couples *(condition, successive step)* which determines the behaviour of the logic flow: if the condition c_{ik} is fulfilled, the step s_i will be followed by the step s_{n_k}, and $1 \leq k \leq n_i$, $1 \leq n_k \leq m$[13].

n_i – the number of conditions (situations) that may appear

S – the set of steps

F – the set of functions

C_i, T - the sets of conditions and types defined below

The "Programming by steps" proposes the set of the steps' types below:

$$T = \{I = Iteration, D = Decision, S = Jump, L = Loop\} \tag{6}$$

Let's see the meaning of each type in relation with the fundamental constructions proposed by the structured programming: sequence, condition and repetition (see section 2.1).

The I type (*iteration*) corresponds to *sequence* construction. If we consider the actual step s_i, the step s_{i+1} will succeed it. There is no condition to be evaluated, and the step number will increase with a fixed iteration, equal to 1 (see Fig. 4a).

The D type (*decision*) corresponds to *decision* construction. The number of conditions to be evaluated is unlimited (like in a multiple selection) and the next step will be calculated based on the condition's evaluation. Let us consider C_i the set of conditions, which are supposed to cover all possibilities, which can arise in the decisional step s_i:

$$C_i = \{c_{ik}\} \qquad 1 \leq i \leq m, \quad 1 \leq k \leq n_i \quad i, k, m, n_i \in N \tag{7}$$

where:

c_{ik} – the condition associated with the step s_i

n_i – the number of possible conditions for the step s_i

m – the number of steps in the flow

If the condition c_{i1} is true, then the step s_{i+1} will follow the step s_i. If the condition c_{i2} is true, then the step s_{i+2} will follow the step s_i, and so on, until the last condition (if the condition c_{in_i} is true, then s_i will be succeeded by s_{i+n_i})[14]. All these branches will meet in the next step s_{i+n_i+1}. The conditions are discrete and finite (see Fig. 4b)[15].

[13]For simplifying, we will consider the relation between the step's index n_k and the condition's index k in the couple (condition, successive step) as linear: $n_k = i + k$. Note that "Programming by steps" is not limited to this linearity.

[14]Remember that when running an application one condition will only be true at a certain moment of time. The flow will pass trough a unique branch.

[15]The problem is to transform a set of continual conditions in a set of discrete conditions. For example, if the condition is: "if x>0 then true else false", then a function will be used in order to translate it like here: f(x) = (unknown char) if x>0 return 1, else return 0ý. The condition becomes: "if f(x)=1 then true, if f(x)=0 then false". We have obtained a discrete condition from a continuous one.

The S type (*jump*) behaves in this manner: the step s_i will be followed by a sequential step s_{i+k}. If k is negative, the logical flow will come back to a precedent step (see Fig. 4c and d). This type doesn't correspond to a fundamental construction (it is similar to the *"go to"* instruction). It offers a lot of flexibility, but it should be used carefully. The number k differs to 1 (otherwise the step's type will be an *iteration* I).

The *repetition* construction has no correspondence to the steps' types, because it can be simulated by a number of iterations, decisions and negative jumps, until the exit condition is fulfilled. This shall be managed carefully, for avoiding infinite repetitions when the application runs with real data. When we use an application for drawing the flow chart, the same information will generate at least the number of cycles foreseen for the real-time situation. Sometimes the exit condition will never happen, because the drawing software simulates the real execution without having the real data. Therefore the number of cycles might be infinite. The L type (*loop*) was introduced in order to solve this issue. It is associated with the last step of a *repetition* in order to mark its end (see Fig. 4e and f). The type behaves as a negative jump[16] and the real application will treat it identically. The drawing application will represent only the first cycle of the repetition.

Based on the description above, the link between the step's type and its behaviour for the step s_i ($1 \leq i \leq m$, $i \in N$) shall be schematised:

- I type – iteration: s_i ->s_{i+1}
- D type – decision: s_i -> - if c_{i1} is true then -> s_{i+1}
  - if c_{i2} is true then -> s_{i+2}
  ... ->s_{i+n_i+1}
  - if c_{in_i} is true then -> s_{i+n_i}
  where $1 \leq i + n_i \leq m$, $n_i \in N$
- L type – loop[17]: s_i ->s_{i+k}, $k < 0$ $k \in N$
- S type – jump: s_i ->s_{i+k}, $k <> 1$ $k \in N$

The Fig. 4 depicts the types' definitions according with the presented method. In this figure the step in discussion is s_i and its type is properly represented. The types of the other steps are irrelevant.

"Programming by steps" considers that *each step is a virtual decision.* A *sequence* is a decision with a unique branch, which is always true. A *repetition* is a decision with two branches: the first branch will return the flow to a precedent step, and the condition will then be re-evaluated, until something changes; the second branch will be followed when the cycle finishes. Only the *decision* has many conditions to be evaluated.

The information related to the sequence of steps (the logical flow) may be presented as in Fig. 5. The table THESE_STEPS contains the information about the current steps: the step's identifier, the step's type, the number[18] of successive steps/conditions and the function's identifier that will be executed at this step.

[16]The iteration and the loop are particular cases of the jump.
[17]Used for repetition only.
[18]It is greater than 1 for a D step, otherwise it is equal to 1.

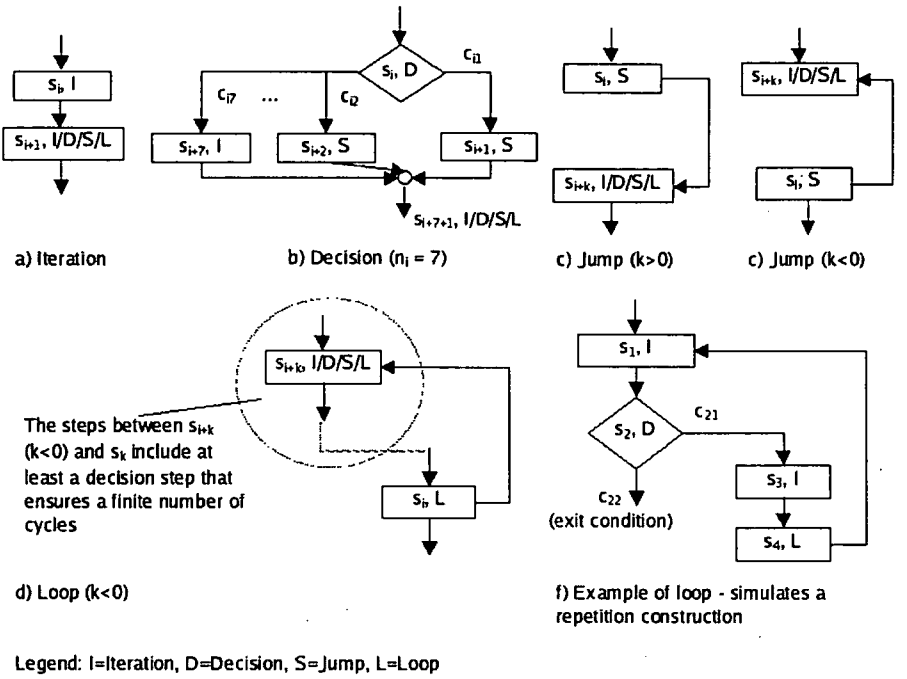


Figure 4: Example of steps' types defined in the "Programming by steps" method

The table NEXT_STEPS is related to the couples (condition, successive step) that force the execution: the current step's identifier, the condition's identifier, the value for which the condition is true and the corresponding next step to be executed (when that condition is realised). For the types which differ from decision, the condition can not be estimated (ID_COND=0, VAL_COND=*<NULL>*). DEF_STEPS contains the set of types, with their description. The main application has implemented a special mechanism in order to automatically treat the different types of steps (see section 3 of this paper).

We can observe that not all the information of the logical flow is memorised, because there is no exchange of data between the functions from one step to another, thus we haven't obtained a functional system yet.

The possible parameters' values have the following characteristics:

- they may be fixed or variable, deterministic or non-deterministic;
- they may be dependent or independent of the step in that are executed, as well as the precedent steps;
- they have a certain type of data and a specific dimension.

To solve the data transfer, "Programming by steps" introduces a new concept: a *field* associated with one or many parameters, which is a part of a specific table:

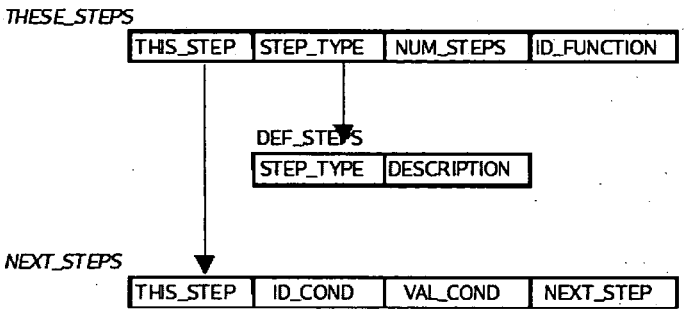


Figure 5: The general structure of the tables with information for the flow building

TRANSFER_VALUES. Transferring data from one step to another means associating the same field with two parameters: one output parameter appertaining to the first function and one input parameter belonging to the successive function (indicated by the specific logical flow). The first parameter will scatter the output value in the field mentioned above, and the second one will gather it to use in its function[19].

For example, we can choose to associate the o_{il} output parameter of the f_i function with the <FIELD_1> field at the s_i step. After we may associate the i_{jk} input parameter of the f_j function with the same field in order to receive the contained data at the s_j step.

The link between the parameters and their fields will be done according to the step where the function is called (see FIELDS_THIS_STEP table). This table contains the step's identifier, the parameter's identifier and the field's identifier (associated with the field's description in FIELDS table).

The TRANSFER_VALUES table is composed of fields having names, types, and initial values as described in FIELDS table. If many applications use the same flow, many rows will be inserted in the table TRANSFER_VALUES, and each application will identify its row by a number (record number). The value memorised into ID_FIELD field (see tables FIELDS, FIELDS_THIS_STEP and FIELDS_NEXT_STEP) indicates the position of the corresponding field in the TRANSFER_VALUES table's structure. Please observe that we may associate now the field <FIELD_1> with other inputs parameter for reusing the value memorised there, or with other output parameters for loading it with a new value.

Let's consider another situation. Supposing the same function is used in the same flow twice, with different values each time. There are two solutions:

- to associate new fields with the respective parameters, memorising their initial

[19]Do you remember the interpretation of the parameters? Let's imagine now the field like a room between two doors (the parameters): the value goes out from the first function and enters in this room using one door. When it is necessary, the other door is open and the value leaves the room, entering in the successive function.

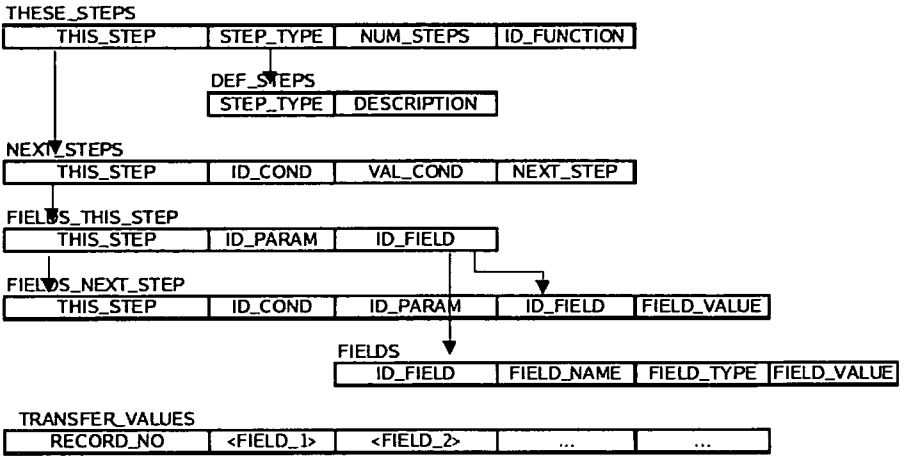


Figure 6: A logical flow implementation in a database management system

values into the FIELD_VALUE fields (see FIELDS table);

- to use the same associated fields and to reset the values.

In the second alternative another data set will be used, that specifies the initialisations to be made for a successive step in a known situation, given by the couple (condition, successive step): FIELDS_NEXT_STEP. The parameters, the fields and the associated values are memorised here, and will be set for a certain condition that is realised during the application's execution. The table contains the step's identifier, the condition's identifier (in order to know the branch), the parameter's identifier, the field's identifier and its initial value.

The "*field*" concept offers a great flexibility in the functions' management. As described above, the fields can be initialised with static values at the beginning or at a certain moment during the run-time. They also permit the data transfer between steps and functions.

Our database schema may be completed like in Fig. 6.

Resuming, we can consider the logical flow as an oriented graph. In each node a certain function is realised and the next step is decided. Parameters' values may be transferred from one node to another, using FIELDS_THIS_STEP table, and some parameters may sometimes be re-initialised, using FIELDS_NEXT_STEP table.

The information related to the step execution may be represented as:

$$s_i = s_i(t_i, \{(c_{i1}, s_{i+1}), \ldots, (c_{ik}, s_{i+k}), \ldots, (c_{in_i}, s_{i+n_i})\}, \\ f_i(v_{i1}, v_{i2}, \ldots, v_{ih}, \ldots, v_{ix_i}, w_{i1}, \ldots, w_{il}, \ldots, w_{iy_i})) \tag{8}$$

$$s_i \in S, \quad t_i \in T, \quad f_i \in F, \quad c_{ik} \in C, \quad s_{i+k} \in S$$
$$1 \leq i, i+k \leq m, \quad 1 \leq k \leq n_i \qquad i, k, m, n_i \in N$$
$$1 \leq h \leq x_i, \quad 1 \leq l \leq y_i \qquad h, l, x_i, y_i \in N$$

where
s_i – the step s_i
t_i – the type of step s_i
f_i – the function specific to the step s_i
v_{ih} – the value s_i of the input parameter i_h specific to the step s_i
w_{il} – the value of the output parameter o_l specific to the step s_i
(c_{ik}, s_{i+k}) – one of the couples (condition, successive step), for step s_i, where $1 \leq k \leq n_i$, $1 \leq i+k \leq m$

2.4 The Main Application

The main application will be built in order to roll over the logical flow memorised in the tables, following the procedure presented in Fig. 7.

In this representation, an iterative alternative was chosen to operate the whole logical flow. The idea to build such a program comes from the backtracking engine used in the nonprocedural language Prolog, that can be stopped only with a specific instruction (in our case: the value of the next step is equal to 0)[20].
In the case of "Programming by steps" method, the application that follows the flow's evolution consists in a "*do-while*" cycle that operates until it receives a specific exit instruction. A flow step is performed at any iteration, so that a specific function is launched, then the next step is identified. The zero value for the step represents the exit from the logical flow (and also from the program).

In this way the logical flows can be built physically, the main program has an role of execution, but is transparent for the content of execution. If the functions have been analysed and projected to be usable in many and different situations and programs, then the programmers work is reduced only to organise the functions in the necessary order and to fill in some linking information in the database tables.

From the Fig. 7 we may deduce the mechanism on which the method "Programming by steps" is based and from where its name comes from: it loads one step by one and executes each associated function. The sequence of steps is linked to the specific characteristics for the problem and for the domain.

3 How to apply "Programming by steps"?

"Programming by steps" method requires the next stages (see Fig. 8):

1. Domain Analysis
2. Main application development

[20]It may be used a recursive method that will be more suggestive when the logical flow is interpreted like an oriented graph.

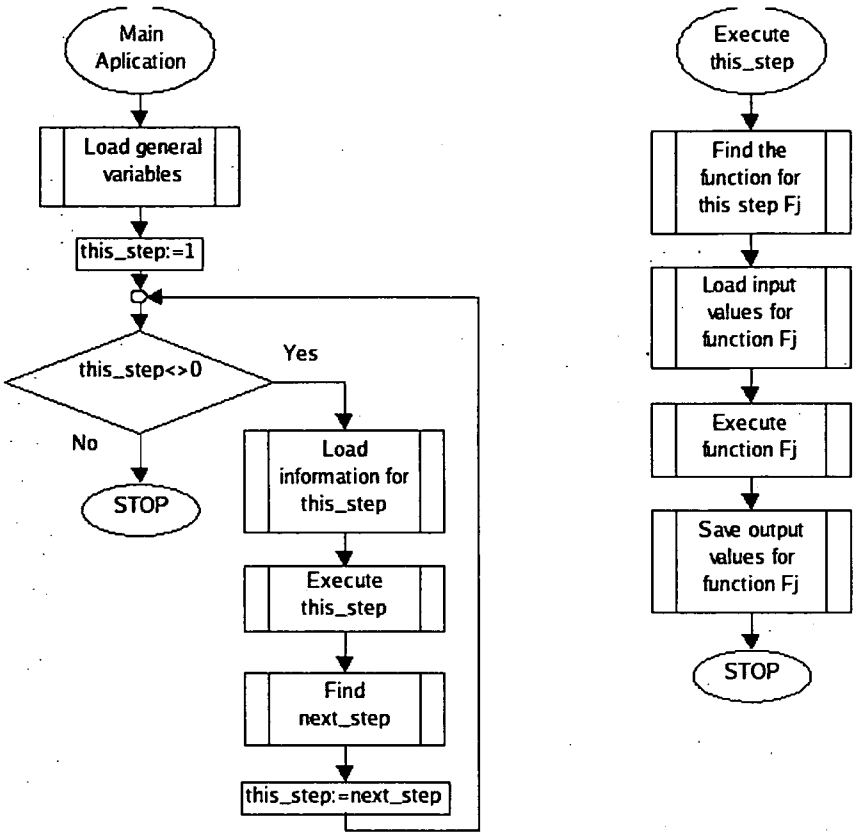


Figure 7: The behaviour of the main application that executes the logical flow

3. Functions development
4. Functions integration in the Common Library
5. Specific problem analysis; if there are new functions to be developed go to 4, else go to 6
6. Flow chart design and implementation for the problem solution
7. Application testing and homologation

For all these stages a short description will be presented[21].

1) *Domain analysis*

[21]Note: More homogeneous functionality has the analysed domain, more simply is to apply the method. If the analysis is correct and the functions present a high degree of standardisation, the steps 1,2,3, and 4 are rarely used.

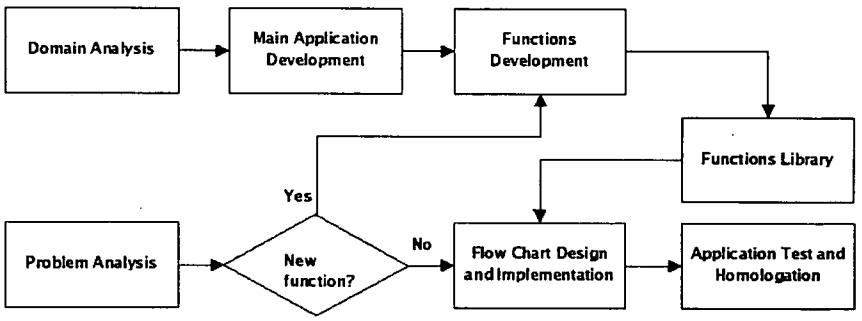


Figure 8: The process model for the method "Programming by steps"

The general description of the domain analysis can be taken over from the oriented objects design method, with the following definition: "the domain analysis for the software consists of the location, analysis and specification of the common requirements in a specific application sector in order to reuse some parts of them in many projects from that sector"[22].

In the "Programming by steps" method, the domain analysis looks for identification of general characteristics of the sector, the location of already existing applications, the identification of future and present requirements, the identification of main requested functions, and the setting up of reusing projecting standards.

2) *Main application development*

This phase permits to build the main application with its interface towards the function library (internally or externally implemented).

The main software reads the information of each step, gathers the input values of the corresponding function and interprets them, runs the function and scatters the output results in associated fields, manages the checking of all the steps, and ensures the execution of the entire system well (see Fig. 7).

The main application's characteristics are defined in the way of interacting with the functions on one side, and with the data on the other side. The modality to execute the steps (recursive or iterative) is chosen, and the engine that follows the logical flow is projected.

In this stage the database structure is implemented, which stores the data of the steps and its priorities, functions and its parameters, static and dynamic values of the parameters, errors etc. The structures that implement these functions are projected.

[22]Unofficial translation from the Italian version of R. S. Pressman in "Principi di ingegneria del software", 2000, pp.598, that refers D. G. Firesmith with "Object Oriented Requirements Analysis and Logical Design", 1993

3) *Functions development*

The analysis starts from the functions identified at the first stage. The input and output parameters are defined in order to be possible to use a function in as many situations as possible.
The name, the data type and its dimension are established for each input/output parameter. In order to raise the usage of a function it is recommended that each parameter accept many data types. Some conversion mechanisms will be applied inside each function.

The data structures defined in second stage and represented in Fig. 3 are loaded. The functions can be written in a high level language, or in the language incorporated in the chosen database management system.

4) *Functions integration in the Common Library*

The role of this stage is to integrate the functions, which were projected at the third stage, into a library. A friendly interface between the main application and the library shall be created in order to reuse the functions if necessary. However, the application will interpret the information memorised at the second stage to access and run the functions.

5) *Specific problem analysis*

In this phase the problem must be identified within the area. It is necessary to find similar problems and their possible solutions. If there is more than one solution, one must be chosen and its corresponding functions must be defined. When all of these functions are defined, we can go directly to the sixth stage.

If the analysis reflects the necessity of new functions, we will return to the third stage – *Functions development,* until all the new functions are integrated into the Common Library.

6) *Flow chart design and implementation for the problem solution*

The projecting and implementation of the logical flow represent the effective programming process. This stage's goal is the structure loading with the specific values for the problem's solution.

It is based on the chosen solution at the fifth point, where the necessary functions were identified and defined, and it continues with the establishing of functions' succession. The logical flow is built up asking questions like: *"What really happens next in the process?", "Does a decision need to be made before the next step?", or "What approvals are required before moving on to the next step?".* The data structures from Fig. 6 are loaded. The initial values are established for each step.

7) *Application testing and homologation*

After putting all the pieces together, the logical flow execution must be tested by the main application. With the flow chart already built, we need to make

a test plan that covers all the different paths and to execute it, correcting the eventual differences from the thought-established solution.

4 Comparative analysis between the "Programming by steps", the "Rapid Prototyping" and the "Component-based Design"

This section contains a short presentation of two other design method (Rapid Prototyping and the Component-based Design) and some comparative characteristics between these methods and the "Programming by steps" method.

The "Rapid Prototyping" method

"The prototyping paradigm may be <closed> or <open>"[23]. In the first case the prototype is considered a "quick and dirty" affair, used like a communications aid between users and developers. Once the sought information has been obtained, it is discarded and conventional software design ensues (see Fig. 9). In the second case, the prototype becomes the central focus of the process model, called "evolutive prototyping". The prototype is scoped, scheduled, resources are allocated and refined as depicted in Fig. 10[24].

"One solution for the "Rapid Prototyping" consists of the prototype assembling (instead of building it), using the existing software components. An existing software product can be used like a prototype for a new product. In a same sense, this is a reuse form applied on the prototyping"[25].

In the case of the "Programming by steps" method, one or many existing applications may constitute a prototype model for a new one. In this way a new problem will be quickly understood and resolved with predictable results. The old functions are all homologated and their use will be secure and free of errors. In case of detected errors, the corrections automatically touch the old applications (logical flows) which therefore makes the maintenance process easier. But the model for a new application is neither a closed prototype, nor an opened one, because it is more, it is a functional model, with correct results and acceptance by the client.

The "Component-based Design" method

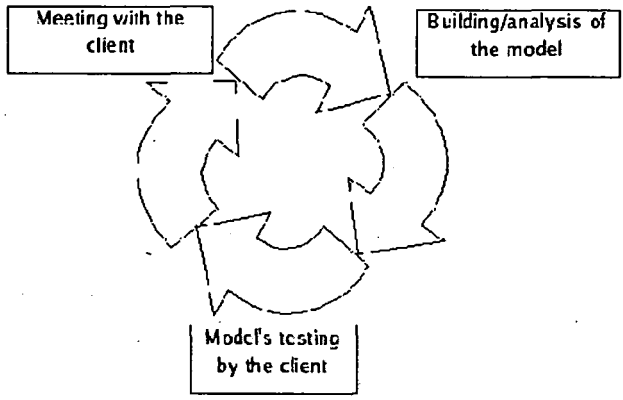
"Component based development offers a vision of plug and play software development"[26]. "The Component-Based Software Engineering process is drastically different from the conventional software development process: it's integration-centric

[23] Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 301

[24] See R. L. Vienneau, R. Senn – "A state of the ART Report: software design methods", 1995, p. 12

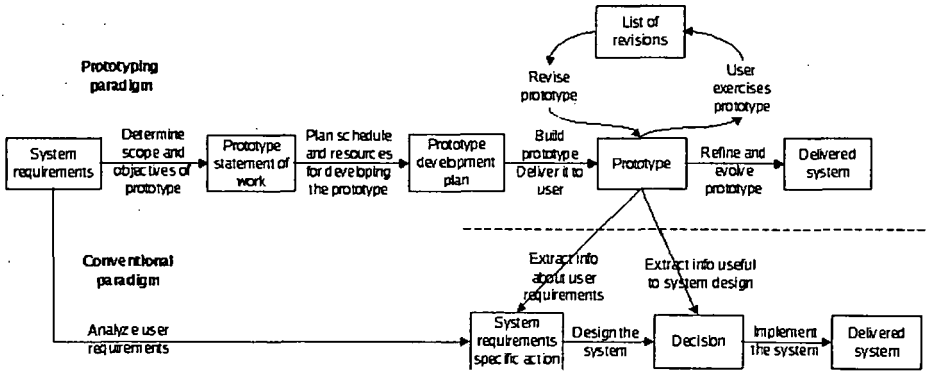
[25] Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 302

[26] See M. Collins-Cope, D. Deveaux, P. Frison, H. Matthews, G. Pour, "Component Based Development: Software Architecture, Component Models and Teaching", p. 1



Source: Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 33

Figure 9: The prototype paradigm



Source: W.W. Agresti, "What are the New Paradigms?"

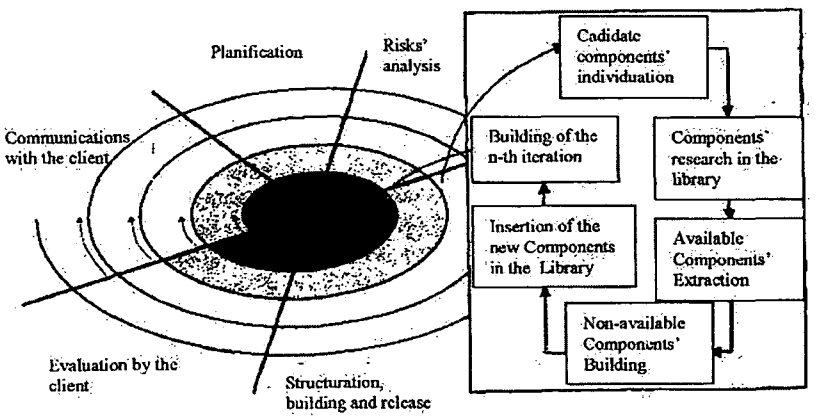
Figure 10: The Prototyping Paradigm and its Relationship to the Conventional Software Development.

as opposed to development-centric. This is a real challenge for developers, but also for teachers. New skills should be emphasised:

- connections between analysis, design and programming,
- documentation use and its construction,

- testing and validation technologies,
- reliability and trustability,
- software project management"[27].

"When building procedural code, common behaviour is extracted into some kind of function or subroutine that can be reused in some way, either by having all users call the same function implementation, or by copying the function into code at compile time.



Source: Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 45

Figure 11: Component-based Development

When designing a component-based solution, it is possible to extract a common behaviour so that multiple consumers can use it"[28].

The "Programming by steps" method may be situated between these two approaches: the common behaviour is extracted into functions like in a procedural method. But, the way in which they are used is nearer to the component-based method, because the main application calls them like external components. The logical flow is composed piece by piece for every solution, reusing the functions. The functions may be used to solve several problems of a specific domain, as well as in a different application domain, if they have the requested functionality and a recognisable interface[29].

[27]See M. Collins-Cope, D. Deveaux, P. Frison, H. Matthews, G. Pour, "Component Based Development: Software Architecture, Component Models and Teaching", p. 1

[28]See K. McInnis, "Component-based Design and Reuse", 1999, p. 2

[29]The problem of CPU's usage shall be solved using a multiprocessor computer.

Table 1: Comparative requirements to apply the design methods: Rapid Prototyping, Component Based Development (CBD) and "Programming by steps"

Requirements	Closed Prototyping	Open Prototyping	CBD	Programming by steps
Application domain well known	*	*	*	*
Problems may be modelled	*	*	*	*
Accurate and stable requirements	-	*	*	*
Ambiguous and contradictory requirements	*	-	-	-
Possibility of reuse	-	*	*	*

Source: adapted from R. S. Pressman, "Principi di ingegneria del software", 2000, p. 302

"The component based development model represented in Fig. 11 incorporates several characteristics of the spiral model. It presents an evolutive nature[30] and requires a software development iterative approach. However this development model creates applications starting from software components ready to be used (the classes)"[31].

We see that also the "Programming by steps" method has an evolutive nature. One could remark on the similarity between the process of component individuation and design on one hand, and the process of function identification and design, on the other hand. One function is like a component, with its functionality and its interface implemented into the database tables (see Fig. 3).

In the next table we may see some requirements that could be used to choose one of the methods:

Different advantages and disadvantages of the three methods are listed in the Table 2.

5 Conclusions

The use of a high level language with a good database management system adds some advantages to the "Programming by steps" method:

- the velocity and accuracy of computing offered by the programming language;

[30]See R. S. Pressman in "Principi di ingegneria del software", 2000, p. 44, that refers the article of Nierstrasz, O., Gibbs, S., Tsichritzis, D., "Component-Oriented Software Development", 1992, pp. 160-165

[31]Unofficial translation from the Italian version of R. S. Pressman, "Principi di ingegneria del software", 2000, p. 44

Table 2: Comparative advantages and disadvantages

Advantages and disadvantages	Rapid Prototyping	CBD	Programming by steps
Rapid understanding of the requirements	*	-	-
Rapid application building	-	*	*
Reuse possibilities	*	*	*
Reducing of the development cycle	-	*	*
Reducing of the project costs	*	*	*
Increasing of the productivity	-	*	*
Possibility to use wizards	*	*	*
Possibility to use non-expert personnel[32]	-	-	*
Prototype may be shallow and narrow	*	-	-
Components too complicated	-	*	-
Integration issues between main application and database implementation	-	-	*

- faster and secure data access and manipulation, without concurrency problems offered by the database management system;
- larger volume of data that could be stored inside of the same database management system.

The idea of the functions' incorporation inside the database management system, and the ability to be updated for different problems without touching the source in the high-level language, raises the flexibility of the program and offers new development's perspectives.

"Programming by steps" is a new method that designs and implements a mechanism that executes specific steps corresponding to a logical flow and permits to build new applications only configuring some tables with the steps and the functions necessary in that case. A new program may be designed building the succession of steps with the necessary functions for the respective situation. In other words, the software engineer would fill in some fields in the tables with some values in order to complete the logical flow and then to run the program.

In a future release a non-specialist user could also be the "writer" of the software. One possible solution is to implement a system, with an adapted graphical interface (for example, an icon for each function). The user will have to choose the icons and to connect them together in a logical flow. Another solution is to use a flow-charting software for drawing the logical flow and then automatically to convert it in the requested information in order to fill in the specific tables.

The "Programming by steps" method is based on the reusable functions that may be designed without changing the main application. That offers flexibility and manageability in obtaining new software releases.

[32]This feature refers to the situation when the library has already been implemented.

References

[1] Agresti, W. W. "What are the New Paradigms?" In Agresti, W.W. (ed.) "New Paradigms for Software Development", Washington, DC: IEEE Computer Society, 1986

[2] Burns, T. "A Fresh Look at Flow Charting", http://www.q-skills.com/flowchrt.html

[3] Chichernea, V., Botezatu, C., Iacob, I., Fabian, C., Mihalcea, R., Goron, S. "Proiectarea sistemelor informatice", Sylvi, Bucharest, 2001

[4] Clauson, J. R., Glenn, T., Hunter, J. A. H. "Index of quality control tutorials", Clemson CQI Server Copyright (c) 1995 by Clemson University Portions Copyright (c), 1995 http://deming.eng.clemson.edu/pub/tutorials/qctools/flowm.htm

[5] Collins-Cope, M., Deveaux, D., Frison, P., Matthews, H., Pour, G. "Component Based Development: Software Architecture, Component Models and Teaching", CBD-TOOL, 2000

[6] Firesmith, D.G. "Object Oriented Requirements Analysis and Logical Design", Wiley, 1993

[7] McInnis, K. "Component-based Design and Reuse", Castek, 1999

[8] Mihalca, R., Tataru, A. "Realizarea produselor program. Metode si tehnici de analiza si proiectare structurata", Scripta, Bucharest, 1994

[9] Nierstrasz, O., Gibbs, S., Tsichritzis, D. "Component-Oriented Software Development", CACM, vol. 35, no. 9, September 1992, pp. 160-165

[10] Pressman, R. S. "Principi di ingegneria del software", third edition, McGraw Hill, Milan, 2000

[11] Vienneau, R. L., Senn, R. "A state of the ART Report: software design methods", ITT System Corporation, Griffiss Bussiness and Technology Park, Rome, 1995 - http://www.dacs.dtic.mil/techs/design/Design.toc.html

[12] *** "The New Oxford Dictionary of English", Clarendon Press, Oxford, 1998

[13] *** ISO 9004-4 "Quality management and quality system elements, Part 4: Guidelines for quality improvement", Geneva, first edition, 1993