

AN OPERATIONAL APPROACH TO PROGRAM DESIGN

P. Rocchi^a and A. Haag^b

^aIBM, via Shangai 53, 00144, Roma, ITALY
paolorocchi@it.ibm.com

^bIBM, via Shangai 53, 00144, Roma, ITALY
alessandro.haag@it.ibm.com

ABSTRACT

All programming languages include declaratives and instructions, but software developers design the latter and neglect the former. In consequence of this default, they improvise the declaratives when they code the program. This article puts forward the field diagram (FD) which completes the program blueprint and provides a significant support to practitioners during the development phase and maintenance.

Keywords: Programming methods, program design, declaratives, field diagram.

1. PRELIMINARY REMARKS

Authors and companies put forward an ample basket of tools for software programming; theoretical assessments clarify the use of some professional aids. Take for example, the Jackson and Warnier approaches to program design, formal methods for development and correctness of algorithms, incremental tests of packages, pre-compilers, test-markers. They prove to support workers and frequently proposals have been adopted with fervor. Nevertheless developers still encounter obstacles in the application development. Difficulties emerge here and there (Blackburn et al. 1996); results fall short of the expectations and frequently do not reach the complete success (Glass 1999).

We believe so many disappointments cannot be accidental.

We note that the construction of software applications includes design, coding, test and maintenance that run nose to tail. The output of the precedent step prompts the next one, thus an inaccuracy or one simple oversight in a phase produces multiple effects in the following stages. We conjecture that a defect emerging in the initial stages could obstruct the software programming.

2. A SHORTCOMING OF SOFTWARE DEVELOPMENT

The engineering design is a preliminary plan or sketch for making something (Koen 1985). A project determines on paper what will be built, so that the result is easily understood, verified and modified. In such a way, engineers smooth the way toward the solution and fulfill their task. A project bridges requirements and production, in particular:

- α) It satisfies the needs of the customer.
- β) It outlines the main parts and the details of the final product.

We wonder if software development goes along with rules α) and β). This article concerns with the technical argument, hence we leave aside the ability of engineers to perceive and translate the customer requirements and focus on point β).

We find the following two parts in a software product:

1. *Instructions* determine the operations that the computer units will execute.
2. *Declaratives* allocate the data-fields in the RAM memory and the peripherals that will run.

All the software techniques revolve around them; structured languages, objects languages, logical languages etc. deal with both instructions and variables (Tucker et al. 2001). Programmers chart the former by using pseudocoding or block diagrams, but do not model the latter. Neither any other plan determines this necessary part as β) demands. Program-designers outline half of the software product and the project is incomplete. No architect charts the first floor of a building while the roof is a fragment of his imagination; instead a software specialist predefines component **1** and keeps the part **2** in mind. Pseudocoding and block diagrams are insufficient but we cannot quote significant remarks about this evident shortage in current literature (Winans 1988). Why do authors neglect this neat shortcoming?

In our opinion, they do not treat this argument as computer science and programming theories are founded upon logical-abstract basis. Theorists refer software programs to Turing's machine which handles immaterial variables and does not operate with fields and peripherals; hence they usually conceive data as ethereal items (Ehrich 1982) and the blueprint of declaratives is taken as unnecessary. Current literature presumes that pseudo-codes and block diagrams should suffice to set up variables. This notion takes hold so strongly that even the authors of the *data driven* methods, i.e. Jackson and Warnier, ignore appropriate models for data and peripherals.

Experiences in the field disprove this simplistic approach. Instructions and declaratives are both essential and none can be omitted during the engineering stage. A case should make this clear. The programmer has two alternative ways to assign a value to a field, for instance he writes this declarative in Pascal:

```
CONST A = 0;
```

or otherwise uses this instruction:

```
A := 0;
```

It is evident that the former and the latter play a symmetrical role and are interchangeable. You may object that the first solution is less significant as some symbolic languages allow implicit declaratives.

We reply that a formal reduction does not entail simplification of contents, to wit implicit declaratives does not minimize the movements of data in the running program. Language's syntax reduces the workload of coding but does not wipe out the relevance of data in between the project. Practice highlights that the development of component **2** as outlined above is demanding and sometimes requires much more effort than instructions do. For example,

- Attributes require accurate checks in o-o programming because a mere detail prevents the class from running.
- Databases and files imply severe declarations to the extent that they may need the support of a specialist.
- The programmer declares several items such as the headline, the generic line, the final line, the page-number etc. with procedural languages.

The program layout stages the first phase in software development. This systematic professional bias frustrates all the ensuing tasks performed by a developer. If any item is lacking or wrong in the instruction design, it cannot be easily checked because of the partial documentation. In short, the layout of declaratives is often lacking and that creates difficulties for

practitioners. It impairs efforts toward effective development processes. We conclude that a suitable scheme should finish the program blueprint.

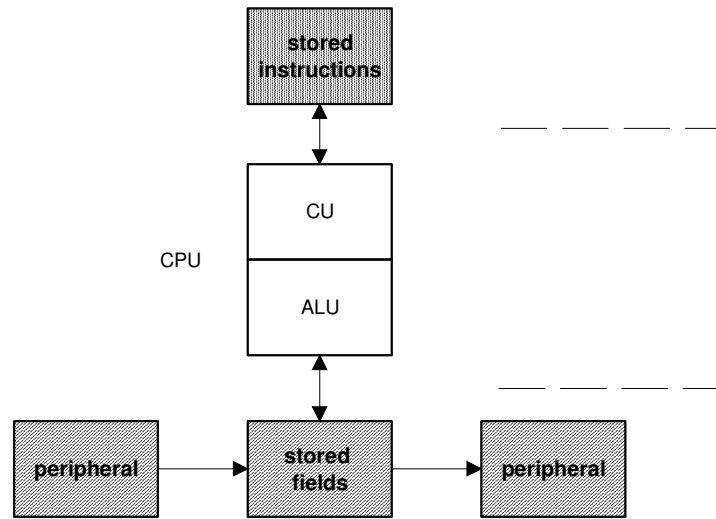


Figure 1. Components 1 and 2 of software programs

3. DESIGN OF DECLARATIVES

We have observed above that the logical-abstract approach to programming averts specialists from the complete design of programs due to the simplified model of the computer system. The correct theoretical reference is necessary to take the right road. We ignore the Turing machine and we base our proposal upon a complete theoretical model, which includes stored instructions and data as well (Randell 1982).

The stored-program computer architecture in Figure 1 complies with the following evident hierarchy: the software instructions determine the work of the control unit that, in turn, affects on ALU and the peripherals. The hierarchical ranks set apart components 1 and 2 and facilitate our search. The bottom of Figure 1 leads us toward the layout of the declaratives. We merely expand the shaded symbols using these graphical rules:

- i) Place traditional symbols of the input/output units at the far left and right in the diagram.
- ii) Draw the fields at the center using segments.
- iii) Place the label (if any) above the field symbol and the initial contents (if any) over the segment.

From now onward, we shall call the graph defined through points i, ii, iii as a *field diagram (FD)*. The term “field” aims at marking the operational and practical viewpoint that we have assumed.

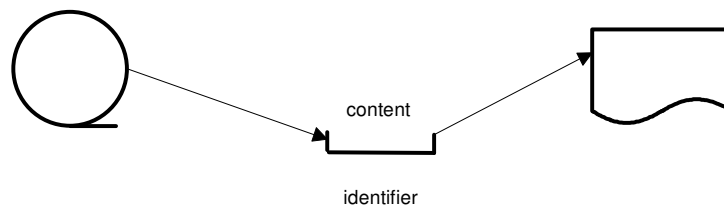


Figure 2. Basics of field diagrams (FD)

A field diagram enables us to design elementary variables and data structures as well. It provides a precise guide to draw record-tracks and areas, attributes of a class, pointers and other items. It even exhibits the registers that do not appear in the declaratives.

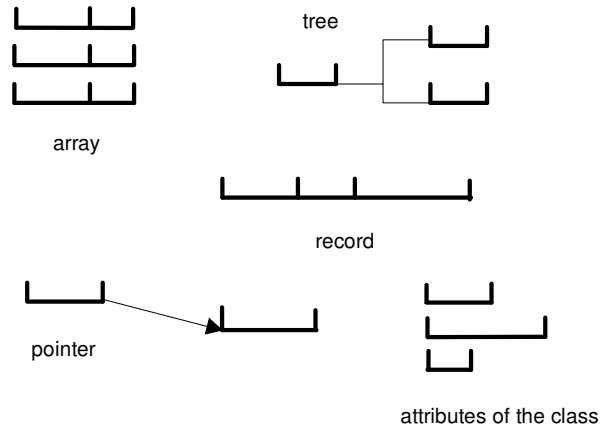


Figure 3. Structures of data shown with FD

Project documentation extends as much as it is necessary. The style, shared by engineers everywhere, accords with this tenet. They expand the blueprint in accordance with the requirements of the work so that they do not waste energy. Unnecessary details are not to be designed in the declaratives layout because they increase the workload of developers. Elasticity emerges as the necessary quality of the approach we are presenting. A field diagram is flexible according to two criteria:

- 1) The field diagram expands or otherwise gets smaller in proportion with the extension of the project.
- 2) The practitioner explodes the passages according to his comprehension of the work. If the logic of the algorithm is intricate, he details the arrows for any instruction. On the other hand, a straightforward logic does not necessitate a FD.

The programmer writes the pseudo-code and tunes a FD in proportion to the obstacles he /she encounters. In practice, a FD proves to be effective in tackling the knotty passages of the algorithm: it aids the developer blocked in front of a conundrum, it reminds details that later will bring difficulties.

For example, let us assume that a Java class instantiates two random variables with the seeds 78 and 36:

```
Random r1 = new Random(78)
Random r2 = new Random(36)
```

Later a method modifies r2

```
r2 = r1;
```

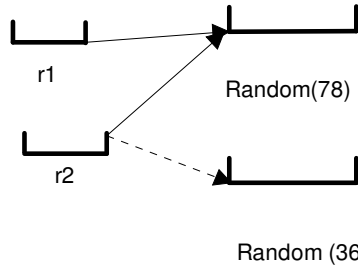


Figure 4. A FD for the preceding example

The field diagram notifies the developer that the garbage collector will cancel the second instance of Random due to the lack of pointers, and Random(36) can no longer be evoked.

The field diagram offers the operational view of the software program and matches with the traditional *flow diagram* (Gremis 1960) and with the *data-flow diagram* of structured analysis (Yourdon 1989). The symbols of a FD agree with the diagrams of Coad for the analysis of classes (Coad and Yourdon 1991). In short it ensures the smooth transition from analysis to coding. A number of misunderstandings between analysts and programmers may be clarified.

4. ADVANTAGES OFFERED BY FD

We comment on the usage of field diagrams in the course of two significant programming phases.

LOGIC: The program prints the name and the age of the user who enters his name and birth-date from the keyboard.

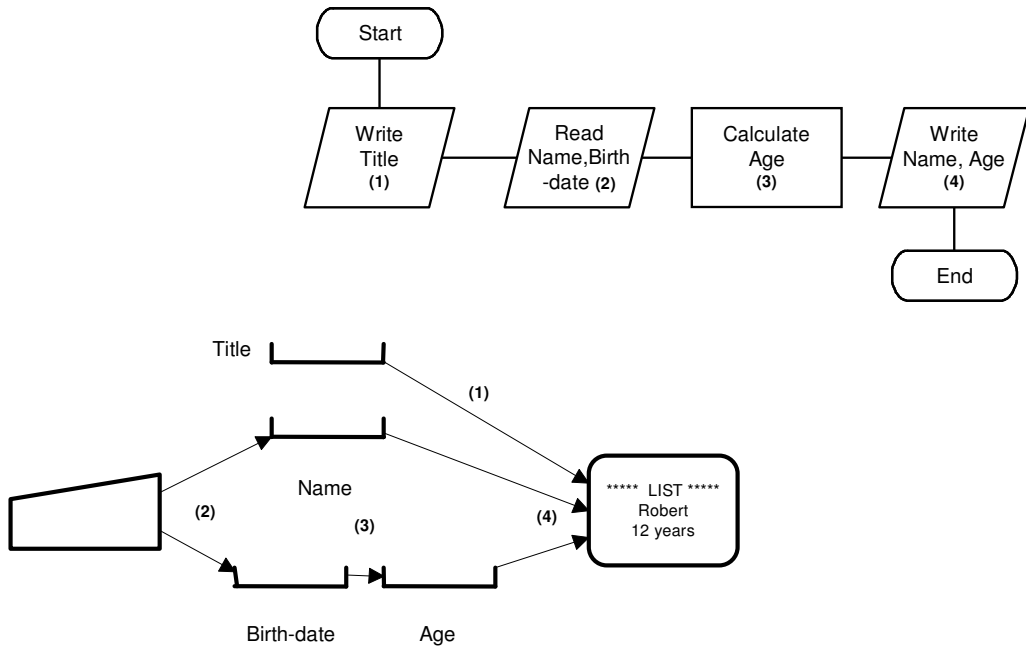


Figure 5. An illustrative case

Development - A computer operation entails movements of data among the fields, FD illustrates these actions and makes explicit what each operation carries out. The developer sees on paper the specific effects of internal and external operations. In particular, the presentation of the instructions and the drawing of the declaratives detail the algorithm from two different and complementary perspectives so that the programmer associates the instruction written in pseudo-code (or block diagram) with the relative arrows of the field diagram. He/she substantiates any item in the former scheme with the latter, and vice versa. Connections are easy because they are visual. The developer crosschecks and immediately establishes if the outcomes, elucidated in the field diagram, comply with the specifications or not. The didactical case presented on Figure 5 illustrates how FD accounts for the flow chart at glance and the pair conforms to the requirements.

Re-engineering and Maintenance - Several programs usually do not have any documentation and their maintenance becomes an awkward task still further. The symbolic instructions suggest pseudocoding that mirrors the viewpoint of the list. Conversely FD elucidates the results step by step and clarifies on paper how the program runs. The practitioner, who has never seen the program before and aims at understanding in brief how the algorithm works, highly regards the qualities of the coupled schemes. To exemplify, we show a Cobol program in the Appendix. The author of that program did not follow the standard and did not use transparent identifiers, thus the result appears rather obscure. The following concise documentation, derived from the list, elucidates how the source program calculates the average of the pressed keys. The numbers of lines within the brackets detail how a FD fits with the code.

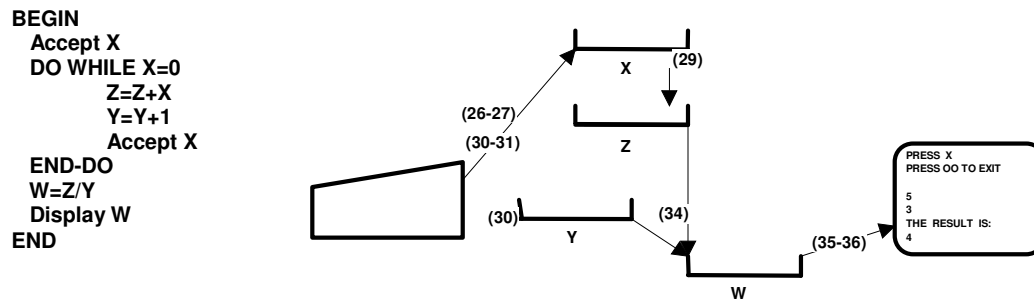


Figure 6. A FD for the program in the appendix

The declaratives' blueprint drawn in parallel with the pseudocoding makes the symbolic program clear and paves the way toward updating. Take for example, the practitioner who is going to recalculate the variable K. He is unaware of the details, notably he ignores how K is really constructed, which data structure embodies the value K, how many instructions manipulate K, if K has influence on other fields etc. The specialist finds difficult to control these facets inside the symbolic program and fears that something may be lost. The field diagram holds special treasures for him; in fact, a FD illustrates all the movements regarding K with the arrows and the practitioner visually checks any particular aspect.

We believe that the coupling of pseudocoding and FDs will confine the endemic problem of program maintenance.

5. CONCLUSIVE REMARKS

Declaratives are as much necessary as instructions, thus the blueprint of declaratives is mandatory. The field diagram aims at perfecting current program design that instead neglects the data fields and the peripherals. A FD matches with pseudocoding and block diagrams and with well-known patterns of structured and object oriented analysis. It meshes with all the tools, methods and approaches currently in use. It may be combined with other schemes such as the transition diagram, Petri nets etc.

The field diagram visualizes the effects that the executable program will produce step by step. It supports the software application with documentary evidence. A programmer easily creates and screens the most intricate functions on paper. In our opinion, the problem of program correctness comes closer to a solution. The complete blueprint, which includes pseudocoding and FD, provides the documentation of the software program without any further investment.

Ample work on software methodologies encompasses the ideas and the results dealt with in this article (Rocchi 2000) and this is the last issue which we aim at highlighting.

REFERENCES

- Blackburn J.D., Scudder G.D., Van Wassenhove L.N. (1996) - Improving Speed and Productivity of Software Development: a Global Survey of Software Developers - *IEEE Transactions on Software Engineering* **22**(12).
- Coad P., Yourdon E. (1991) - *Object Oriented Design* - Yourdon Press, Englewood Cliffs, N.J.
- Ehrich H. D. (1982) - On the Theory of Specification of Abstract Data Types – *J. of the Association for Computing Machinery*, **29**(1).
- Glass R. L. (1999) - *Computing Calamities: Lessons Learned From Products Projects and Companies that Failed* - Prentice-Hall.
- Grems M. (1960) - Shared Standard Flow Chart Symbols - *Communications of the ACM*, **3**(3).
- Koen B.V. (1985) - *Definition of the Engineering Method* - American Society for Engineering Education.
- Randell B. (ed.) (1982) - *The Origins of Digital Computers , Selected Papers* - Springer-Verlag, N.Y.
- Rocchi P. (2000) - *Technology + Culture = Software* - IOS Press, Amsterdam.
- Tucker A.B., Noonan R.E., Noonan R. (2001) - *Programming Languages: Principles and Paradigms* - McGraw-Hill, N.Y.
- Winans R.T. (1988) - Claudius, an Interactive Graphical Software Design Method – *Proc. of the Twentieth Southeastern Symposium on System Theory*.
- Yourdon E. (1989) - *Modern Structured Analysis* - Yourdon Press, N.Y.

APPENDIX AN ILLUSTRATIVE PROGRAM IN COBOL

```

000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. CES3.
000003 ENVIRONMENT DIVISION.
000004 CONFIGURATION SECTION.
000005     SPECIAL-NAMES.
000006     DECIMAL-POINT IS COMMA.
000007 DATA DIVISION.
000008 WORKING-STORAGE SECTION.
000009     01 X PIC 9(10).
000010     01 Y PIC 9(10) VALUE 0.
000011     01 Z PIC 9(10) VALUE 0.
000012     01 W PIC 9(10)V99 VALUE 0.
000013 SCREEN SECTION.
000014     01 VID1.
000015         02 BLANK SCREEN FOREGROUND-COLOR 3 BACKGROUND-COLOR 1.
000016         02 LINE 15 COLUMN 20 VALUE 'PRESS X'.
000017         02 LINE 15 COLUMN 40 VALUE '(PRESS 00 TO EXIT)'.
000018         02 LINE 15 COLUMN 65 PIC 9(10) TO X.
000019     01 VID2.
000020         02 BLANK SCREEN FOREGROUND-COLOR 3 BACKGROUND-COLOR 1.
000021         02 LINE 15 COLUMN 20 VALUE 'THE RESULT'.
000022         02 LINE 15 COLUMN 34 PIC Z(10),99 FROM W.
000023     01 VID3.
000024         02 LINE 20 COLUMN 40 VALUE 'END !!!'.
000025 PROCEDURE DIVISION.
000026     DISPLAY VID1.
000027     ACCEPT VID1.

```

```
000028      PERFORM UNTIL X IS ZERO
000029          COMPUTE Z = Z + X
000030          ADD 1 TO Y
000031          DISPLAY VID1
000032          ACCEPT VID1
000033      END-PERFORM.
000034      COMPUTE W = Z / Y.
000035      DISPLAY VID2.
000036      DISPLAY VID3.
          37          STOP-RUN.
```

Received: 11 August 2003.

Accepted in final form: 30 April 2004 after two revisions.

About the authors:

P. Rocchi and A. Haag are researchers at IBM, Italy. Letters can be addressed to them at IBM, via Shangai 53, 00144, Roma, ITALY. You can reach them also by email at paolorocchi@it.ibm.com and alessandro.haag@it.ibm.com .