SLAC-PUB-2456
January 1980
(M)

THE BABEL OF APPLICATION DEVELOPMENT TOOLS*

John R. Ehrman
Manager, User Services
SLAC Computing Services (Mail Bin 97)
Stanford Linear Accelerator Center
Stanford, California 94305

ABSTRACT

A major barrier to achieving productivity in the development of all programming applications is the poor quality and excessive number of tools that must be used to get the job done. Most attempts to find, measure, and eliminate productivity bottlenecks have focused on only one of these tools, namely higher-level languages. An example of such an attempt is the recent concern with Structured Programming.

It is the thesis of this presentation that in order to produce even a simple program, a programmer must know on the order of a dozen distinct languages. These languages are incompatible, have idiosyncratic syntaxes and mediocre diagnostics, and are often the biggest stumbling blocks in the way of rapid program development. A simple programming example is used to illustrate these points.

(Submitted to Datamation Magazine)

* Supported by the Department of Energy under contract DE-AC03-76SF00515.

(Originally prepared for presentation at the SHARE-GUIDE-IBM Application Development Symposium, Monterey, CA., October 15 - 17, 1979.)

1. OVERVIEW

1.1 INTRODUCTION

It is well known today that the rapidly falling cost of computer hardware is making the increasing cost of software more and more prominent as a factor in the development and maintenance of computing applications. Hence, increasing the usability and useful lifetime of programs and the productivity of programmers have become major concerns throughout the data processing industry.

I contend that we have done little to enhance the productivity of programming personnel, or to help them improve the quality of what is produced. In fact, we seem to be making the job of producing long-lasting application code harder even as we try to simplify it. While this may seem to be a contradiction in terms, I believe I can demonstrate how and why this situation is occurring, and what can be done to alleviate it.

1. One significant barrier to increased productivity is that an "average" application programmer must know on the order of a dozen different languages in order to create and maintain an application. The number of such languages is not decreasing.

2. I believe the problem will continue to get worse. The widespread use of micro-programmed devices means that the number of "language-like" interfaces is increasing, and that they have an ever greater variety of styles.

3. The growing number of user interfaces is making it harder to combine new and old application programs to form larger and more general ones.

4. We must no longer optimize the usage of computer resources at the expense of human resources.

I believe these problems might be alleviated by a change in the way we view the application process: we should eliminate the artificial languages and replace them with a subset of normal English.

In addition to its normal meaning, I will use the term "language" to mean any interface between a computer and its user that requires the user to learn and understand the peculiarities and details of transactions across that interface.

I will illustrate these points by following a typical program development process through a series of steps.

2. AN ILLUSTRATION OF THE PROBLEM

2.1 THE PROGRAM DEVELOPMENT PROCESS

Suppose you assigned one of your programmers (whom we will call Flo Coder) to the task of implementing a new system to maintain a file of data relating to (say) an inventory of parts important to your business. The programmer was to devise the structure of the file containing the data, the layout of the reports to be produced from the data file, the organization of the program, and so forth. Let us follow her on an imaginary path leading toward (not necessarily to) the solution to the problem.

2.2 THE PROBLEM DEFINITION

First, Flo sat down with the people who requested the program be written, and derived a detailed specification of all the requirements the program was intended to satisfy. Since the program was expected to be small, and the programming effort limited to one person, the specifications could be written out and understood in a relatively small number of pages. (This is in marked contrast to some large software projects involving hundreds of programmers, analysts, and supervisors: planning and controlling such large projects requires large and complex computer-based systems just to help specify and track the phases of the job. This example is purposely kept simple!)

After looking over the problem specifications and analyzing them thoroughly, Flo wrote out a set of hierarchically-organized, top-down structured-programming data flow and HIPO diagrams.

2.3 THE PROGRAMMING LANGUAGE

Flo chose to write the program in PL/I, since it was the most general and flexible high-level language available. Part of the program is shown here:

```
    IF (NUM_REMAINING <= REORDER_LEVEL) THEN
       DO; PUT SKIP EDIT FILE(REORDERS)
               (PART_NUMBER, PART_DESCRIP, ORDER_COUNT)
               (    F(20),       A(40),        F(12)   ) ;
           REORDER_COUNT = REORDER_COUNT + 1 ;
       END;
```

Figure 1: Part of a Program

In this example, there are actually two distinct programming languages being used. The first is the language of statement flow and process sequencing, the "logical organization" of the code. (This first language is the one that has been thoroughly scrutinized by devotees of Structured Programming. You will observe that there are no GOTO statements in the example!)

The second language is what we might call a "formatting and conversion language". The two middle lines are needed to specify how the program's internal data must be rearranged and converted before it can be put onto an external file. Another example of this second language appears in lines 3 and 4 of the part of the program where data is to be read into the program:

```
 WHILE (END_FILE = FALSE)
   DO ; GET EDIT (PART_NUMBER, PART_DESCRIP, PART_COUNT,
               NUM_REMAINING, REORDER_LEVEL, ORDER_COUNT)
              (SKIP, F(10), A(32), (4) F(8) ) ;
   END ;
```

Figure 2: Another Part of a Program

As in Figure 1, part of the code in this example involves the mapping of data from its representations and organization on an external medium into the format and organization it will have internal to the program.

There is a third language that must be used to specify the <u>internal</u> data types and structures to be manipulated by the processing logic of the program:

```
    DECLARE (PART_NUMBER, PART_COUNT, NUM_REMAINING,
             REORDER_LEVEL, ORDER_COUNT) FIXED BINARY(31),
             PART_DESCRIP CHARACTER(32) ;
```

Figure 3: Declaring Internal Variables

These three examples illustrate the fact that writing a program requires knowledge not only of an algorithmic language that manipulates the internal data in a prescribed fashion, but also a knowledge of the external formats and representations of all data elements, <u>and</u> the special "format conversion" language that transforms the data between its internal and external formats and structures. Thus our programmer Flo had to know three distinct languages just to get the program "down on paper":

L1. an algorithmic data-manipulation and statement-sequencing language;
L2. a "format conversion" language specifying the structure of the external data, and the mapping between the external organization and representation of data and its internal organization and representation;
L3. a language for specifying the internal structure of the data elements to be manipulated by the algorithm.

Figure 4: A High-Level Language is 3 Distinct Languages

2.4 <u>JOB CONTROL LANGUAGE</u>

Flo compiled her program and got rid of the errors that the compiler detected. To try to debug the program by actually executing it, Flo had to tell the Operating System what she wanted done. This required the use of control statements such as Job Control Language (JCL):

```
//FLOTEST1  JOB  'F.CODER,2631',TIME=(1,30)
//TESTCASE  EXEC PL1XCLG,
//               PARM.PL1='STMT,NEST,LIST,ETC',
//               PARM.LKED='RENT,LIST,XREF',
//               REGION.GO=256K
//PL1.SYSIN     DD DISP=SHR,DSN=DEPT23.E2631.CODER.TEST1
//LKED.SYSLMOD DD DISP=OLD,DSN=DEPT23.E2631.CODER.LMOD
//GO.INPUT      DD DISP=SHR,DSN=DEPT23.E2631.CODER.DATA
```

Figure 5: JCL Statements

JCL is hard to learn even in its "pure" form; in addition, each installation imposes local JCL conventions and requirements. Learning how to use and code these JCL statements correctly requires a substantial investment of time and effort. Flo wrote the JCL statements she needed to get her task started (not done!). To be able to do this, she had to learn both a new language (JCL), and a new set of concepts, to understand the objects to which the control language refers: the Operating System, its structure, and its many components.

L4. Job Control Language, the concepts and facilities to to which it refers, and its "Messages and Codes".

Figure 6: Another Language: JCL

(A note: I happen to have chosen examples from IBM manuals or systems only because I know them best; I'm sure you can find equivalent examples on your system!)

2.5 PROGRAM LINKING

Having gotten the program accepted by the compiler, Flo was ready to linkage-edit it into a test library from which it could be run with some sample data. She found that it was necessary to use some Linkage Editor control statements in order to obtain the results she desired, somewhat like the following:

```
        IDENTIFY  ('INVENTORY-A4/FLO CODER/VERO,MOD2')
        INCLUDE   SYSTEM(MODULEA,MODULEB)
        INCLUDE   UPDATE(FIXTOC)
        ORDER     MAIN,A,B,C,D,E,F
        ALIAS     INVENT
        SETSSI    ABCDEF01
        NAME      INVENTO(R)
```

Figure 7: Linkage Editor Input Statements

Unfortunately, she got some diagnostics from the Linkage Editor that required getting some help from Joe Veal, the systems expert next door:

```
 ****INVENTO   DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
 ****INVENT    IS AN ALIAS FOR THIS MEMBER
 AUTHORIZATION CODE IS       0.
 **MODULE HAS BEEN MARKED NOT REENTERABLE, AND NOT REUSABLE.
```

Figure 8: Linkage Editor Output

Joe explained the way the Operating System handles such matters as Load Modules, reusability characteristics, and what is really meant when the Linkage Editor says that something "DOES NOT EXIST". Thus, we find that Flo has mastered yet another language:

L5. The properties, command syntax, and diagnostics of the Linkage Editor, as well as the concepts and notation for describing the storage and manipulation of programs in load module format.

Figure 9: The Linkage Editor Language

2.6 DEBUGGING AND DIAGNOSTIC TOOLS

Having created a workable version of the program, Flo was ready to test it with some sample data. Due to various oversights and omissions, the program "blew up" in unanticipated ways and produced a dump of memory. In order to find the cause, Flo had to correlate the symbolic Assembler-Language-like listing produced by the compiler with the hexadecimal machine language code and data in the dump. Once she had localized the problem to a small area of the program, she added some extra tests to the code to try to expose the error condition. Fortunately, she could use an interactive debugging system that allowed her to trace the execution of the program on a statement-by-statement basis.

Thus, we find that program debugging requires a knowledge of three additional languages:

L6. Absolute binary machine language.
L7. The symbolic Assembler Language for the machine, and the conventions of that language that apply to the program's execution environment.
L8. The syntax and semantics of the debugging language and the debug_control system.

Figure 10: Debug and Diagnostic Languages

While some progress has been made in providing programmers with better diagnostic facilities for high-level languages, too much diagnostic information is still produced in the form of memory dumps. (IBM's PL/I actually has a robust run-time environment that rarely dumps. Fortran's is less solid, and COBOL's is rather wobbly, so on average this example isn't too unfair.)

2.7 UTILITIES

In order to set up different test versions of her programs and data, Flo had to use a variety of utility programs. Each of these utilities has its own peculiar control statement syntax and format. For example, IEBCOPY requires the usual mix of commas, parentheses, and equal signs, but allows freer format than JCL:

```
     COPY  OUTDD=INOUTA
                INDD=INOUTE
     SELECT  MEMBER=MA,MJ
     COPY  O=INOUTB,I=((INOUTC,R),INOUTD)
     SELECT  MEMBER=((B,H),(C,J,R),A,(D,K))
```

Figure 11: Sample IEBCOPY Control Statements

Similarly, the VSAM Access Method Services language is oriented toward ease of computer scanning, not for ease of use; its designer preferred blanks, lots of parentheses, and hyphens:

```
    REPRO  INFILE (SORT -
             ENV (RECORDFORMAT (F) -
                  BLOCKSIZE (80) -
                  PDEV (3330) ) ) -
           OUTFILE (NAME)
```

Figure 12: Sample VSAM Access Method Services Input

I have actually made a generous grouping of the many distinct and often very dissimilar "utility" languages a programmer must use to complete even a relatively straightforward task. On a typical OS system, a user needs to know how to use IEBGENER, IEBCOMPR, IEBPTPCH, IEBCOPY, IEHLIST, IEHMOVE, IEHPROGM, and IMBLIST, in addition to locally-written utility programs.

Learning the functions of these programs, the objects they manipulate, and the control statements and JCL to obtain needed results, requires another substantial investment of time and effort.

L9. The properties, command syntax, and diagnostics of the Utilities, and the concepts and structure of the objects they manipulate.

Figure 13: The "Utilities" Languages

So far, Flo had to use nine distinct languages, and there was still more to be done!

2.8 TEXT EDITING

An important program development tool is the text editor. In fact, it is probably <u>the</u> most important tool, since it is needed to manipulate the "source" (character) form of <u>all</u> the other objects. Because it is so fundamental to all programming tasks, it should be the easiest to use. It often is as difficult to use as any of the other tools; most programmers simply adapt because it must be so frequently used.

```
top#alter ¬ af * *
dstring/weather/
getfile double items c 10 25
change /_+// 1 *
```

Figure 14: Typical Editor Commands

Flo's computing installation was typical of many others in offering half a dozen or more editors, each with different features, different command syntaxes, and different or inconsistent command and operand names. Furthermore, these editors use a wide variety of internal and external representations for the text: the records may be fixed or variable length, sequence numbers may or may not be part of the record, the text may be a stream of characters with embedded delimiters, and so forth.

L10. A set of commands for manipulating data in character format, and the internal and external representations of that data.

Figure 15: The Text Editor Language

2.9 COMMAND PROCEDURES

Most of today's operating systems provide some simple form of command language procedure capability that allows users to collect and combine commonly-used command strings into a single grouping. Examples are JCL cataloged procedures, TSO's CLIST facility, and the CMS EXEC facility. Note that this language uses ampersands to denote keywords, left parentheses, and no commas:

```
&LNAME = &CONCAT &1 *
LISTFILE &LNAME SCRIPT * (EXEC
EXEC CMS &STACK
&LOOP -END &READFLAG EQ CONSOLE
&READ VARS &NAME &TYPE &MOD
&SUFFIX = &SUBSTR &NAME 3 6
&NEWNAM = &CONCAT &2 &SUFFIX
&IF &RETCODE EQ 0 &SKIP
```

Figure 16: Segment of a CMS EXEC

Most system command languages are so difficult to use without command procedures that it is essentially mandatory to know the command procedure language as well.

L11. A language for writing command sequences and for controlling the sequencing of a set of programs.

Figure 17: Command Procedure Language

2.10 DOCUMENTATION

Flo finally arrived at the end of her programming task: the programs ran correctly, they were upgraded to include specification changes, and the results were acceptable to the users. All that remained to do was to complete the documentation.

However, Flo's program documentation was to be computer-formatted, so she also had to learn the rudiments of a text-formatting language. A typical language used for formatting documents is SCRIPT; note how it differs in style from all the preceding examples:

```
.se pubTFnum = '&&&1'
.ur .pt &pubTFnum.           &2
.pt .sp c;.dh set 1
.ur .sr temp = L'&pubTFnum
.ur .if L'&temp lt &temp1;.th .fo centre
.el .ur .of L'&pubFIhdr+L'&pubTFnum+4
```

Figure 18: Sample of SCRIPT Language

L12. Documentation is produced with the help of a text formatting language.

Figure 19: The Text-Formatting Language

2.11 PLAIN ENGLISH

Unfortunately, writing the documentation often turns out to be the hardest task of all! In addition to all the other languages Flo must master to get the job done, she is expected to be skilled in the use of "Plain English"!

L13. Well-written documentation in Plain English is needed to explain the uses of the product, to aid in marketing it, and to tell how it can be maintained.

Figure 20: The Ultimate Language: "Plain English"

Well-written documentation can save a lot of user support effort and field representatives; if the user can't understand the manuals, he calls on the supplier for help!

Needless to say, it isn't easy to find people who can write clear English in addition to their skills in using many artificial computer languages.

2.12 OTHER LANGUAGES

Our examples of programming languages drawn from Flo's simple programming task did not require some of the additional languages often needed in other applications. In more realistic situations, programmers must handle other problems as well.

1. Different techniques and utility programs are necessary to maintain multiple versions of source, object, and

executable code, along with the patches and temporary fixes at each level.

2. Special languages are used for stating problem requirements and task design specifications, and for project monitoring and control.

3. There are special languages used for describing and accessing the contents of data bases, and for describing the interfaces from other programs into a data base system.

4. Reports and statistical analyses are customarily produced with specialized report-generating and data-analysis languages.

5. Last and by no means least, every programmer must plow through piles of incomplete, inconsistent, inadequate, and even incorrect documentation to try to learn all of the truly <u>necessary</u> languages.

Learning to program well requires learning each of these many languages well, and knowing which is appropriate under which circumstances.

Other languages often needed include:

* Code maintenance systems
* Problem and requirements specifications
* Data base description and controls
* Statistical and report generation
* Plotting and Graphics

Figure 21: Other Languages Often Needed

2.13 SUMMARY

To summarize, our fable of Flo Coder shows that development of even the simplest applications requires knowledge of a round dozen languages.

L1. Algorithmic "processing-logic," statement flow
L2. External data description and conversion
L3. Internal data typing and structuring
L4. JCL or its equivalent
L5. Linkage Editor or Loader
L6. Absolute binary machine language
L7. Symbolic Assembler Language
L8. Debug and Diagnostic System
L9. Utilities
L10. Text Editor
L11. Command procedures
L12. Text Formatter
L13. Plain English

Figure 22: The "Basic Dozen" Languages

These artificial languages were designed to simplify the mechanics of translating them into actions for the machine to obey. Programmers are forced to learn a variety of unnatural languages to do their work. And, because the managers of such programmers typically do not have the same levels of knowledge of these artificial languages, communication between manager and programmer is made more difficult.

Each of these programming tools has its own particular syntax, semantics, and data objects. There is little or no compatibility among the syntaxes, semantics, and data objects of each of the languages. The diagnostics of each language are couched in terms peculiar to the language and its area of application; little of what is learned in one area can be carried over to another.

There is a further difficulty with these artificial languages. As we move from lower to higher levels of control, the languages change. For example, we have

1. programming languages to manipulate data,

2. command languages to control programs,

3. command procedures to control commands,

4. text editors to manipulate the others.

These languages are all different from one another, which makes it very hard to combine procedures at different levels into one coherent application package. To do this now, a programmer must write "scaffolding" or "bridging software" with the sole function of holding the pieces together.

I know of only one major exception to this situation: APL. I strongly suspect that its popularity is due in no small measure to the fact that only one language is needed to use an APL system effectively!

The artificial languages programmers must use are:

* mutually incompatible in syntax
* inconsistent in terminology
* uninformative in diagnostics
* difficult to learn and explain
* designed for computers, not for people
* different from one level of control to the next

Figure 23: Problems with Artificial Languages

3. WHERE WE ARE TODAY

A programmer is required to be a polyglot, in much the same sense as a United Nations translator: he must be able to interpret the sense and intent of a problem statement, and translate it into a dozen other languages with as little loss of meaning as possible. And, like real translators, the skill and effort required often leave little time for reflections upon the deeper meanings of the process itself.

Another effect of having to program in so many different languages is that programmers must constantly "reinvent the wheel." A function coded in one language must be recoded in another because there is so little ability to "mix and match" among languages.

Is it any wonder that a programmer may tend to become fascinated with computing abstractions for their own sake, or that he may occasionally lose sight of his organization's objectives?

3.1 PROLIFERATING INTERFACES

The increasing use of microprogrammed devices means that the number of people designing end-user interfaces is also increasing. Thus, the number of "languages" a user must know will increase as well. To put it bluntly, anybody can create a new jargon, dialect, or entire language, and thereby force other people to "speak" it if they must use the facilities his device or system provides.

More microprogrammed devices

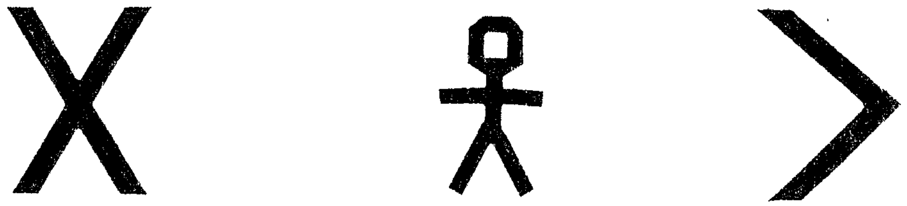
mean

More user-interface languages

Figure 24: Language Proliferation

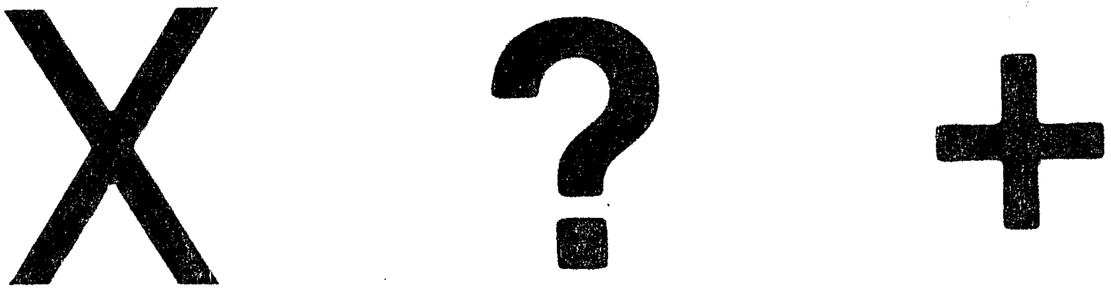
People have tried various ways to deal with this proliferation. For example, language standards activities are obviously driven by the desperate need to reduce the number of different "tongues" and "dialects". Another attempt is Structured Programming, in which intelligent use of only one of the many existing languages (the language of statement flow and process sequencing, called L1 above) was expected to "solve" the problem. (The fact that positive results appeared can probably be attributed as much to a computer-style "Hawthorne Effect" as to any actual benefits.) There has also been considerable research into ways of structuring internal and external data more reliably; again, this involves only one of the many languages programmers must know and use.

On the hardware side, the example of the IBM 3278 terminal is instructive. There is an area at the bottom of the screen where little hieroglyphics are displayed to convey status and error conditions. However, the design of the terminal also includes a little flip-open drawer containing a long, skinny spiral-bound booklet that explains what the hieroglyphs mean! The effort and cost that went into designing both the display and the booklet could easily have been spent on some simple, accurate, and terse English-language explanations of the error conditions.



Meaning is: "TOO MUCH DATA"

Figure 25: IBM 3278 Terminal Diagnostic



Meaning is: "WHAT YOU ENTERED DID NOT GET TO THE HOST"

Figure 26: Another 3278 Terminal Diagnostic

The multiplicity of new and linguistically non-compatible systems at the hardware level (System/1, System/6, System/38, System/370, 4341, 8100, etc.) exacerbates the problem, because the greater variety of low-level architectures and technology encourages an even greater multiplicity of higher level software interfaces.

3.2 REQUIREMENTS

It is clear what is needed:

1. We must reduce the confusing multiplicity of languages and syntaxes needed to develop even the simplest applications.

2. We must minimize the number of translations from the original English-language specifications into artificial languages.

3. There need not be a single language to solve all problems, but a single language should be sufficient to solve a large set of problems. (Again, the example of APL is instructive!)

4. Linguistic consistency and uniformity is needed to be able to combine procedures written at different levels of procedural control.

4. A PROPOSAL

In every case we have examined, the specifications of the task to be done must be translated from the programmer's "natural" language, English (or other native tongue), into an artificial language. In that process, much of the original information content is in danger of being lost. It is natural to propose that no translation be needed.

I am proposing that essentially all programming tasks be done in a carefully restricted subset of English, organized in a way that can be parsed, interpreted, and understood with a minimum of extra training in the concepts of this restricted subset.

That is, we need to devise a language that can be mapped without ambiguity into machine "directives," whether it is compiled into machine code, translated into an internal form and interpreted, or interpreted in its source form.

Such a standard syntax and grammar has one major advantage: all problem-oriented terminology can be mapped into a standard "canonical form" that can then be interpreted or compiled (or a mixture of each) in a way that is architecture-independent.

We will also need some standard rules for creating "shorthands," "jargons," and other forms of abbreviation, because every profession develops terse and convenient ways of expressing common ideas with greater economy.

It is important to observe that this proposal is exactly the inverse of the current situation. We now take the problem description, written in a language natural to people, and translate it into terse artificial languages that can be easily mapped into machine code. I am proposing that we treat English as the language "closest to the machine." Then, if they like, users can create terse "jargons" that abbreviate the full form of the language into more convenient notations. Thus, the fundamental form of an algorithm or procedure should be its expression in the subset of English, not its most terse and machine-digestible representation.

There are numerous advantages to choosing English as the language in which applications are to be written.

1. Applications can be developed by their end users.

   a) There will be less reliance on programmers, who do not understand the needs of the application environment as well as the end users.

   b) The number of potential application program creators will be far greater.

   c) The application program will be under the control of its end users; therefore, it can grow to meet organizational needs as those needs change.

2. The application development process will be easier to teach and learn.

3. Applications can be expected to have much longer lifetimes, because they will be easier to modify and expand as needed.

4. Applications will have more portability from one architecture to another.

5. The market for computing equipment will be broadened.

In short, I am proposing a tool-making tool that will provide end users with enough power and flexibility to produce their <u>own</u> application programs without the necessary intermediation of a "programmer".