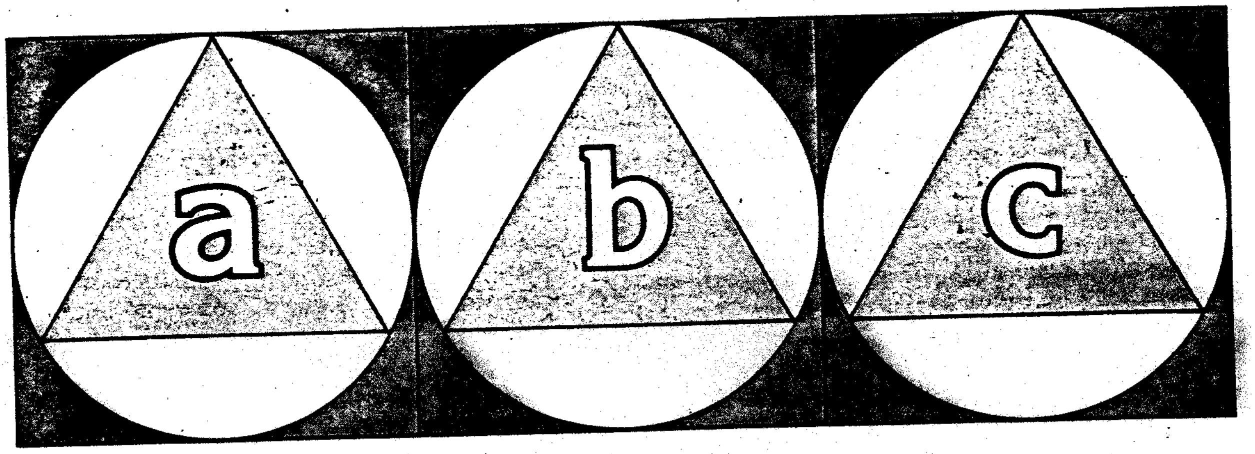
An experimental design language gives evidence that software design may benefit from the same formal techniques programmers are using.



Special Feature:

The Flex Software Design System: Designers Need Languages, Too

Stephen A. Sutton, Digital Technology Incorporated
Victor R. Basili, University of Maryland

First, there were programmers. They wrote programs that told computers what to do and how to do it. As programming systems grew to fill ballooning machine capacity, they became unwieldy, disorganized, and expensive. It became easier to rebuild a program from scratch than to change it. Some people began to seek a means of controlling these systems in their upper, abstract representations. These people became known as software system designers.

Designers needed and developed their own special tools for expressing and manipulating system structure. Just as programmers had developed programming languages, designers sought "design languages." However, while a programmer communicates with the computer, a designer communicates with people: managers, programmers, other designers, and himself. The language requirements are therefore different.

Design languages: a design tool. A design language captures the essential elements of a software design in a form that can easily be understood. Jensen and Tonies[1] give an excellent definition, which we paraphrase:

Design languages are a clear, concise communication medium. Their purpose is twofold: (1) to help designers communicate by identifying commonly understood terms and concepts, and (2) to capture the design decisions in a machine-processable form (as distinguished from a machine-executable form).

Once the software design has been expressed in a machine-processable form, many programs can be created to provide information about it or manipulate it. These programs can: (1) generate design documents in special formats, (2) ensure that the design follows certain design principles, (3) automatically test and/or evaluate the design by simulating its execution, and (4) manage the components of the design as they advance through the development process.

Several existing programming tools could be classified as design languages. They bear a variety of names and embody varying philosophies of system design. Some are based on existing programming languages; others are only remotely related to programming languages. Some give detailed descriptions of procedures, while others yield only the global structure of a software design.

At their simplest, design languages are loosely defined, informal, and not automated. In this form they are often called pseudocode, metacode, pencil-and-paper language, or pidgin English. At their most ambitious level, they have the detail of a programming language and employ processors that are much like modern compilers. At this level they have come to be known as process (or program) design languages, or PDLs.[2-6] Systems that support the specification of software systems[7-9] have much in common with design languages.

0018-9162/81/1100-0095$00.75 © 1981 IEEE

A PDL should have the following traits:

- Multilevel application. It should be useful at all levels of design, from early conception to detailed procedural description.
- Naturalness. Its form and substance should be familiar to designers and programmers. It should include a language element for stating an action or condition in some other formal notation, or in "plain English."
- Flexibility. It should be adapted to the software environment—not vice versa.
- Currency. It should support the latest software methodology.
- Open-endedness. It should accept new concepts and serve as a testbed for them.
- Automated support. The design should exist in a machine-readable form. The design system should include not only the PDL but a processor to check the PDL representation of the design for consistency and the utilities necessary to support the design effort. If not automated, the design of a software system can suffer the same problems as the design process itself: inconsistency in person-to-person communication.
- Management aid and control function. Finally, the design-language system should allow managers to control and evaluate both the design and the design activity.

Programming languages are not for design. In general, modern programming languages do not have these features. Although many could be used as pencil-and-paper design languages, their compilers are created for translation to machine code—they do not support the pure design effort. Except for some "extensible" features, they are based on a standard syntax, not an adaptable one.

The *concepts* in the newer programming languages, however, are important to design languages. Modularity, type abstraction, and "structured" control statements all originated in programming languages. The diffusion of these concepts through the programming community can be slow, simply because access to the computers on which the new languages run is limited. Design languages are far more portable and can more readily promote such new ideas.

Winograd[10] believes that advanced programming systems should do far more than those of the past. The systems he describes have much in common with design systems: "The main goal of a programming system should be to provide a uniform framework for the information that now appears in the declarations, assertions, and documentation. The detailed specification of executable instructions is a secondary activity, and the language should not be distorted to emphasize it. The system should provide a set of tools for generating, manipulating, and integrating descriptions of both results and processes. The activity that we think of as 'writing a program' is only one part of the overall activity that the system must support, and emphasis should be given to understanding rather than creating programs."

Flex: the PDL generator

The Flex design system is a design language and its processor. It combines features found in PDLs with those of modern programming languages into a system that can be adapted to many design environments. There is by no means a consensus of opinion on what a PDL should look like. Flex was designed as an experimental tool that can be of practical use in real software development situations and whose form may change in response to experience and new ideas. Flex was developed on the Prime 400 computer system at the Naval Research Laboratory in Washington, DC. It is being installed on the Univac 1108 system at the University of Maryland and on a VAX 11/780 system.

A flexible language. In both form and function, the Flex language has much in common with modern programming languages.[11-14] It is modular, extensible, and strongly typed. Flex is more adaptable to particular design environments than are programming languages; the form of the basic Flex language is easily extended, restricted, or changed. Its features include

- Control of element interaction: who can do what to which other parts of the design.
- Local name scopes: design groups can use their own internal identifiers without conflicting with those of other groups.
- Definable types: stacks, lists, queues, associative memories, etc.
- Fully generic routines: routines that perform a common function for several data types, such as INITIALIZE and PRINT.
- Definable operators: arithmetic, boolean, string, etc.
- "Stuctured" statements and expressions.

Flex is more detailed and more automated than previous PDLs. It can be configured to look and perform like a number of simpler, less flexible design languages. The full Flex language has a detailed syntax, and its documentation[15,16] can overwhelm the casual user. However, the language can be condensed to simpler forms. For example, it could be configured as a strucured Fortran processor whose form would be readily understood and translated by Fortran programmers. It could have the standard Fortran data types and subroutine structure but offer interface type checking, controlled access to common, structured statements, etc.

A language overview

A programming system is the solution to a particular software design problem. It is complete in that every element (data base, routine, etc.) referenced from within the system is also contained in the system.

The programming system consists of a set of modules, each of which in turn is made up of a set of segments. There are three kinds of segments: data segments, definition segments, and routines. The module is the principal scoping concept in Flex. It provides a local, protected atmosphere for its segments. It can be used to encapsulate a data base or a type definition, or simply to provide a logical partition of the programming system. Data segments contain shared data that can be accessed by routines. Definition segments contain type and operator definitions.

There are two kinds of generic, recursive routines: functions, which are value-returning, and procedures, which are not. In addition, there are two special types of functions: access fuctions, which return by reference (as opposed to return by value), and iteration functions, which define the manner in which the elements of a defined data structure are to be traversed during an iteration loop. Routines are not block structured; they cannot be nested as they can in many Algol-based languages. Instead, they follow a modular structure similar to that of CLU[14] and Simpl.[17]

Access to data. Routines can request access to data residing in data segments in the programming system but not to internal data in other routines. They can promise not to change the data segment (FIX access) or may declare their intention to alter it (ALT access).

Data segments may include other data segments as if they were a part of themselves. This gives rise to an *access tree* in which access to the data segment at any node also allows access to all of its descendants. Routines and data segments can also gain access to definition segments in order to declare variables of the types defined within the segments.

Routines may be declared as FIX, ALT, or CLOSED to determine the manner in which they interact with their environment. ALT routines are free to alter any data to which they have ALT access. FIX routines may alter only their own internal data. CLOSED routines may alter no permanent data and therefore have no history dependence—their action depends only upon their parameters.

Comments and escapes. Comments and escapes are the natural language element in Flex. Comments are used as in programming languages; they provide commentary on the surrounding text. Escapes are similar in form, but are used in place of statements, expressions, type specifications, etc. They are the major vehicle for top-down design, where the natural language description of the early design is successively replaced by more concrete ones until a detailed, procedural description is reached.

Types and operators. The language provides few types. Users define new types with the *parameterized type macro*, which is similar to that found in other type-abstraction languages.[18] The module can be used to encapsulate a type definition: all routines that are allowed to "see" the internal structure of variables defined to be of a certain type are placed in the same module as the type definition. The user must also define all infix and prefix operators and the routines they represent. Flex is extensible in that selected modules containing these definitions can be made global to the programming system. Their information is automatically available to each segment in the design and looks as if it were built into the language.

Structured statements. Flex has a small set of structured statements that can be modified or extended. In particular, the iteration (loop) statements are modular and can quickly be configured for a particular design environment, according to the tastes of the designers.

Global checking. The processor maintains a global view of the programming system and can therefore ensure consistency among remotely separated elements of the design. The design can be processed at all levels. The early design may be largely in natural language (escapes), and the processor will mainly check access restrictions (who has what kind of access to what). As concrete parts of the design appear—that is, as escapes are replaced by detailed statements—they can be checked immediately for consistency. Errors can be detected and corrected before the design is expanded any further.

A hierarchy of users

There are three cooperating users of the Flex system: the *caretaker*, the *administrator*, and the *designer*. Each defines the environment for the ones "below" him—the caretaker for the administrator and designer, the administrator for the designer. Figure 1 shows how each of these users modifies the basic language. The programmer translates the design solution into machine-executable code.

The caretaker builds the skeleton. The caretaker is the custodian of the processor software, controlling the syntax and semantics of the language. The caretaker may be called upon to implement the following typical features, which range from easy to moderate in difficulty (a few manhours to a few manweeks):

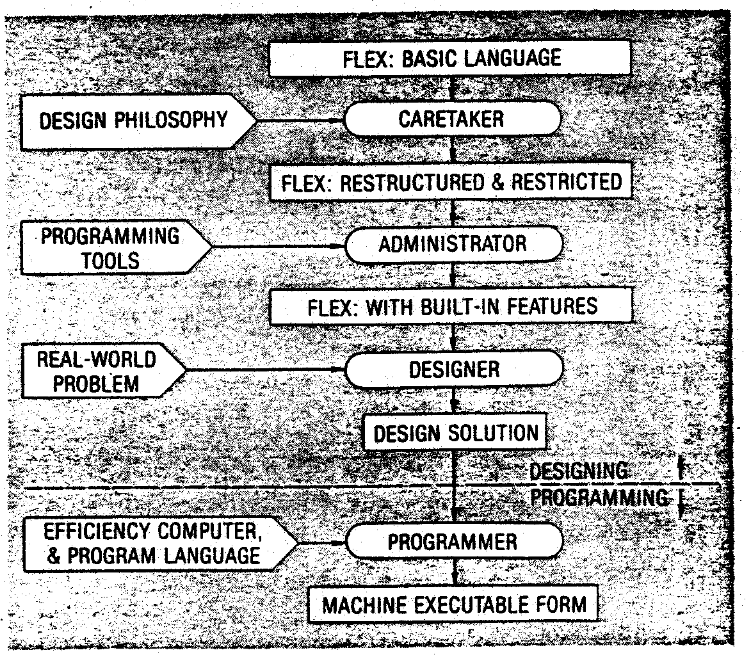


Figure 1. A hierarchy of users.

- New reserved words, or simple syntax changes,
- Special-purpose, natural language declarations in modules or segments—for example:

"PURPOSE:	[text]"	to state its purpose.
"TARGET DATE:	[text]"	to project a completion date.
"PROGRESS:	[text]"	to briefly state its progress.

- New statements, such as one of the many iteration loops that have been proposed in the open literature,
- Special software metrics, such as measures of complexity,
- Restrictions on the way segments interact,
- Assertions—boolean statements that must be true over regions of a routine,
- Exception handling, and
- Routine-valued variables, with type checking.

```
1   closed proc FIND (fix TARGET, fix TABLE, alt LOC)
2     form TARGET string              [[string to be found]]
3     form TABLE  list (string)       [[a table of strings]]
3     form LOC    int                 [[location of TARGET in TABLE,
                                        or 0 if not found]]

4              [search for and return the first occurrence
5               of TARGET in the list TABLE]

6   corp
```

Figure 2. A procedure in the basic Flex language.

```
 1  procedure FIND

 2    purpose: [[Find the position of a string in a list]]

 3    attrib:      closed

 4    input:       TARGET string        [[string to be found]]
 5                 TABLE list (string)  [[list to be searched]]

 6    output:      LOC integer          [[position of TARGET in TABLE,
 7                                        or 0 if not found]]

 8    at exit:     ((LOC ge 0) and (LOC le LENGTH (TABLE)))  and
 9                 (if LOC = 0 then
10                   foreach X in TABLE
11                     not (TABLE (X) = TARGET)
12                   endforeach
13                  else
14                   TABLE (LOC) = TARGET
15                  fi)

16    note:        [[finds only first occurrence of TARGET]]
17                 [[lower case chars are not = to upper case]]
18                 [[if TARGET is the null string,
                     LOC will be nonzero only if
                     TABLE contains the null string]]

19    method:      [[simple linear search from front of TABLE]]

20    body:        [[filled in later]]

21  end procedure
```

Figure 3. Revised FIND procedure.

The caretaker can influence or even define the design philosophy by adding features that enforce certain design practices. He can encourage the use of top-down design techniques by creating different versions of the language for the different stages of the design. For example, he may not allow statements within routines in the early stages, or the changing of interface definitions in later ones.

The administrator fills in the flesh. The administrator uses the extensible features of the language version from the caretaker to build a set of programming tools for the designer. The administrator will usually provide the following language features:

- Arithmetic and boolean operations and their operators,
- Relational functions, equivalence and their operators,
- The assignment procedure and its operator,
- Common data structures and their operations: stacks, lists, strings, arrays, etc., and
- Iteration functions for these data structures.

The Flex language is strongly type checked, but the generic routine structure allows certain automatic type conversions to be defined. For example, the administrator must define equivalence and assignment. He must decide whether to require the operands to be the same type, or to allow certain implicit conversions—for example, real to integer.

Common operations created by the administrator can apply to several data structures. Assignment and equivalence could be defined for any two data structures of the same type; addition could be defined for arrays or lists.

Generic routines can appear to the designer as though they were built into the language. A procedure "INIT" could be created to indicate that a data object of any type is to be set to some initial state (set integers and reals to zero, stacks and queues to empty, etc.). A procedure PRINT could indicate that a data object of any type is to be printed to some I/O device according to some format.

Designers use the result. The designer uses a language that may have been tailored to his particular application. Like the administrator, he may be able to further extend the language, or he may have been restricted by the caretaker to an inflexible language. The designer's language may look like a particular programming language to ease the translation into that language.

Examples

In the examples that follow, escapes and comments are enclosed in single and double brackets, respectively. The boxes around the examples and the line numbers to the left are for reference; they are not part of the language.

Changing the form. Figure 2 shows a procedure written in the basic Flex language. It accepts a TARGET string and a TABLE (list) of strings, and returns the position of the target string in the table.

The basic Flex form is succinct, and, as in programming languages, little is required that is solely for human understanding. The emphasis is on automated consistency checking. This form may be unappealing to some users. The caretaker could devise an alternate form (Figure 3) with little change to the meaning of the routine.

The alternate form resembles statements in a specification language. It includes special clauses to encourage the designer to state how the routine works and the assumptions it makes. The detailed procedural description is of little interest, at least at this level of design. The only added substance is the "at exit:" clause (line 8). This is a boolean expression that could be checked at run time just before exit from the routine or that could be used in formal proofs.

Information hiding. Parnas[19] introduced the term "information hiding" to describe the process of limiting the knowledge of the internal representation of a process or structure to a special group of routines. Type abstraction and encapsulation are examples of information hiding. The caretaker could modify the rules of the basic Flex language to enforce information hiding in special modules.

In Figure 4, a symbol table that might be found in a language translator is hidden in a special table module with a set of routines that alone are allowed to access it. The following restrictions are to be imposed on a table module:

(1) Data segments cannot be exported; no one outside the module may reference its data.
(2) The data segment that holds the main data must have the same name as the module as an aid to the reader.
(3) Routines within the module cannot depend upon shared data outside the module.

To fulfill the last condition, the caretaker must add a new language feature: the MCLOSED routine. This is similar to a CLOSED routine except that it may depend on (i.e., read or write) shared data inside its own module but not on shared data outside its module. This ensures that any values it returns will depend only on the parameters it is passed and on data stored inside the module.

Defining assignment. In Flex, assignment is simply a procedure with two parameters. A symbolic operator (e.g., ":=") can be defined to call the procedure. Figure 5 is typical of the way the administrator may define assignment. This definition will usually be made global to the programming system, as if it were built into the language. The definition segment defines the assignment operator. Whenever a statement is encountered of the form

$$X := Y$$

the processor will issue a call to ASGMOD:ASSIGN, as if the statement had been given as

call ASGMOD:ASSIGN (alt *X*, fix *Y*)

The types of the formal parameters (lines 8 and 9) state that the type of the first can be any type at all, but the second must be of the same type; assignment is allowed between any two objects of the same type. The procedure is CLOSED to ensure that the assignment depends only upon the parameters and that it cannot be influenced by anything else.

Other assignment-like procedures could be defined—for example, an "exchange" statement where the following statement causes two data objects to exchange values:

$$X :: Y$$

The administrator may decide to allow integer types to be assigned to real types, and vice versa. Figure 6 shows a revised assignment procedure.

Routines in Flex may have several cases. The first that matches the interface types in a particular call is the one activated. The first case in Figure 6 (lines 2-14) handles the assignment of real to integer types, the second handles assignment of integers to reals in similar fashion, and the third case will "catch" all other assignment types.

In this example, the administrator explicitly stated the actions to be taken when the value of the real variable ex-

```
table mod SYMTAB

   export FIND, APPEND   [[let outsiders call these]]
-----------------------------------------------------
   data SYMTAB
     [[all data of the symbol table]]
   atad
-----------------------------------------------------
   mclosed func FIND (NAME)
     use fix SYMTAB     [[must read the symbol table]]
       [find a named entry in the symbol table]
   cnuf
-----------------------------------------------------
   mclosed proc APPEND (NAME, VALUE)
     use alt SYMTAB     [[must write to symbol table]!
       [append a new name/value pair to the table]
   corp
dom
```

Figure 4. A "Table" module for information hiding.

```
 1  mod ASGMOD

 2     export fix ASSIGN__OP           [[let outsiders use ':=' operation]]
 3     export ASSIGN                   [[let outsiders call ASSIGN]]
---------------------------------------------------------------------------
 4     def ASSIGN__OP                  [[def symbolic infix operation]]
 5       infix ':=' = proc ASGMOD:ASSIGN
 6     fed
---------------------------------------------------------------------------
 7     closed proc ASSIGN (alt DEST, fix SORC)
 8       form DEST unbound             [[destination of the assignment]]
 9       form SORC typeof (DEST)       [[the source]]

10       [copy SORC into DEST]
11     corp
12  dom
```

Figure 5. Typical definition of an assignment procedure.

```
closed proc ASSIGN (alt DEST, fix SORC)

    case
      form DEST int
      form SORC real
        if [SORC is larger than max integer] then
             [set DEST to largest integer value]
             [send a warning to user]

        elseif [SORC is smaller than smallest neg int] then
             [set DEST to smallest (largest negative) integer]
             [send a warning to user]

        else
             [round off SORC and assign to DEST]

        fi
    esac
----------------------------------------------------------------
    case
      form DEST real
      form SORC int
           [[similar to first case, except
             treat conversion from int to real]]
    esac
----------------------------------------------------------------
    case
           [[same as Figure 6]]
    esac
corp
```

Figure 6. Alternate form for assignment.

ceeds that of the integer precision of the computer. The implementer would then know exactly what he must do to handle these situations.

Who benefits?

The design teams. Design teams using Flex benefit because everyone is speaking the same language. Although modules may exist in different versions, they all stem from the same root, and the variation can be controlled.

Designers also benefit because the process is automated. The Flex processor detects inconsistencies (interface errors, FIX/ALT access violations, etc.), and design walkthroughs can concentrate on whether or not the design is complete, correct, and satisfies the specifications.

With the aid of a librarian, a centralized design data base can be maintained and controlled. To aid this process, the processor produces a map of the programming system with a cross-reference listing, and the caretaker might modify the processor to provide other indices into the design. For example, he might require the "PURPOSE: [text]" declaration at the beginning of each routine, and the processor could then list these for all routines in the design.

The managers. The processor has features that a manager can use to determine the progress of the design. It keeps several size counts for each module (number of lines, comments, escapes, nonblank characters, nonblank characters in comments, etc.). The manager can determine which modules are lagging and the extent to which modules are being documented. The ratio of escapes to other statements gives some idea of the progress of the top-down design.

Software engineering researchers. A flexible automated design language can be used as a testbed for software engineering methods and theories. The Flex processor generates data on the syntactic structure of a design, and the caretaker could add other specific measures that could be used to study features of the design process. Design groups could be given different versions of the language, and their resulting progress, or lack of it, would test the value of different design methods.

Documentation. Design language text can be self-documenting (as can the text of a program). Since there are different levels of the design text, there are different levels of documentation, each providing only the detail needed for its level. The caretaker may add features to encourage documentation—for example, a special comment following each data declaration, as shown in Figure 3.

What are the disadvantages?

Many features of a design language that are abstractly pleasing (such as the assignment of large data structures) can be quite inefficient if translated literally into a target programming language. The programmers must then give special attention to rearranging for efficiency, and the target code may have a structure radically different from the design.

Because the design text is never executed, features whose faults are not obvious until run time occur only in the target language. They must be "back translated" into the design language and cured. It may even be that these faults are corrected, perhaps unknowingly, when translated into the target language.

There is overhead associated with maintaining a design as well as a programming language data base that small organizations may not be able to absorb.

Flex has its own drawbacks. Many of the interesting modifications to the Flex system require the attention of a caretaker who has more-than-casual familiarity with the processor software. The design team will need to designate a caretaker in much the same way as a librarian is designated.

The Flex processor does a lot of processing and is relatively slow. But just as an ounce of prevention is worth a pound of cure, an error caught early in the design process can avoid expensive changes later.

Over the horizon

We want to avoid the "basket weaving" syndrome: building a tool solely for the sake of building it, rather than for the benefit of its users. The most important task of the Flex system is to prove its own value.

There are precedents, however, that attest to the value of automated design systems. Caine and Gordon[3] and Van Leer[4] both describe PDLs that are less ambitious than Flex. They are both enthusiastic about the success of these tools in production environments. Automated design languages can provide a library design data base similar to the program libraries that Brooks[20] describes as "one of the best-done things in the OS/360 effort."

Support tools. There are several extensions to the Flex system. Many of these tools depend upon the specific syntax of the language being used, however, and loss of flexibility must be considered.

Interactive query systems. Programs that interactively query the design could answer specific questions from managers or designers. The information would be current, freshly drawn from the actual design. Other programs could present graphs of the programming system.

Special editors. Special text editors would help to input the design and better control access to it. They could preprocess parts of the design and keep track of which parts had recently been changed and needed to be reprocessed. They could provide templates similar to the one in Figure 3 and let the designers "fill in the blanks."

Specification systems. Specification, the statement of the problem that a software system is to solve, is generally not addressed by design languages. Although a specification and its software solution should not be structurally related (the person who states the problem should only state needs, not specify how those needs are to be met), there may be a way of extending the general concepts of design languages to accommodate specification. For example, Figure 3 is similar to the PSL/PSA specification language.[7]

Simulators. When a particular language syntax becomes standard in an environment, an interpreter could be written to execute the design. The run-time environment can be thoroughly controlled with features too expensive to be included in production programming languages. The interpreter can be as machine independent as the design language itself.

The last word

The Flex software design system gives the software designer a flexible language with which to communicate his designs, as well as providing him with the precise, machine-checkable form and function of a modern programming language. The system has benefits for designers, program managers, and programmers. Since this tool is relatively new, there are many questions as to how it will perform in both experimental and practical environments. Its early use will be aimed at answering these questions. ■

Acknowledgment

The bulk of this work was carried out at the Naval Research Laboratory, Materials Science Division, Washington, DC.

References

1. R. W. Jensen and C. C. Tonies, *Software Engineering*, Prentice-Hall, Englewood Cliffs, N.J., 1978, p. 97.
2. R. C. Linger, H. D. Mills, and B. Witt, *Structured Programming Theory and Practice*, Addison-Wesley, Reading, Mass., 1977.
3. S. H. Caine and E. K. Gordon, "PDL—A Tool for Software Design," *AFIPS Conf. Proc.*, Vol. 44, 1975 NCC, pp. 271-276.
4. P. Van Leer, "Top-Down-Development Using a Program Design Language," *IBM Systems J.*, Vol. 15, No. 2, 1976, pp. 155-170.
5. R. Boyd, and A. Pizzarello, "Introduction to the WELLMADE Design Methodology," *IEEE Trans. Software Eng.*, Vol. SE-4, No. 8, July 1978, pp. 276-282.
6. B. P. Buckles, "Formal Module Level Specifications," *Proc. ACM Ann. Conf.*, Seattle, Wash., Oct. 1977, pp. 138-144.
7. D. Teichroew, and E. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 41-48.
8. D. T. Ross and K. E. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 6-15.
9. T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 49-60.
10. T. Winograd, "Beyond Programming Languages," *Comm. ACM*, Vol. 22, No. 7, July 1979, pp. 391-401.
11. B. Wegbreit, "The Treatment of Data Types in EL1," *Comm. ACM*, Vol. 17, No. 5, May 1974, pp. 251-264.
12. K. Jensen and N. Wirth, *Pascal—User Manual Report*, Springer-Verlag, New York, 1974.
13. B. W. Lampson et al., "Report on the Programming Language EUCLID," *SIGPLAN Notices* (ACM), Vol. 12, No. 2, Feb. 1977, pp. 1-79.
14. B. Liskov et al., "Abstraction Mechanisms in CLU," *Comm. ACM*, Vol. 20, No. 8, Aug. 1977, pp. 564-576.
15. S. A. Sutton, and V. R. Basili, "FLEX: A Flexible, Automated Process Design System," NRL Report 8349, Naval Research Laboratory, Washington, DC, 1979.
16. S. A. Sutton, "The FLEX System: User and Caretaker's Manual," Technical Report TR-765, Department of Computer Science, University of Maryland, 1979.
17. V. R. Basili, and A. J. Turner, *SIMPL-T: A Structured Programming Language*, Paladin House, Geneva, Ill., 1976.
18. J. B. Morris, "A Synopsis of Data Type Abstraction in Programming Languages," Los Alamos Scientific Laboratory Report LA-UR-76-1750, 1976.
19. D. L. Parnas, "On the Criteria Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.
20. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1974.

Stephen A. Sutton is a senior software designer and quality assurance administrator at Digital Technology, Inc., in Champaign, Illinois. His professional interests include all areas of software engineering and software quality assurance, particularly software testing and verification. Previously, while at the Naval Research Laboratory in Washington, DC, he designed and implemented computer systems for automated mechanical testing of advanced military materials.

Sutton received his BS and MS degrees in theoretical and applied mechanics from the University of Illinois in 1973. He received an MS in computer science from the University of Maryland in 1979.

Victor R. Basili has been a member of the faculty of the Department of Computer Science at the University of Maryland since 1970, where he is presently an associate professor. From 1963 to 1967 he taught in the mathematics and computer science department at Providence College, Providence, Rhode Island. He has been involved in the design and development of several software projects, including the Simpl family of structured programming languages, the graph algorithmic language, Graal, and the SL/1 language for the CDC Star. Currently he is involved in the measurement and evaluation of software development at the NASA/Goddard Space Flight Center.

Basili has acted as a consultant for several industrial organizations and government agencies, including IBM, GE, CSC, NRL, NSWC, and NASA. He is a member of ACM, the IEEE Computer Society, and the American Association of University Professors. He received his BS degree in mathematics from Fordham College, his MS in mathematics from Syracuse University, and his PhD in computer science from the University of Texas at Austin, in 1961, 1963, and 1970, respectively.

Computer Professionals

In today's job market you can most likely pick up the telephone, call a friend or aquaintance and have job interviews arranged in 48 hours! But is this really the best way to approach such an important process? At Winter, Wyman we don't believe it is.

Our business is the business of contacts, and our staff of professional consultants can provide you with objective evaluations which will help you choose among job alternatives the one that will best satisfy your personal and professional goals.

With Winter Wyman, you can confidentially explore career opportunities aligned with your interests and get the advantage of our knowledge of the marketplace and wide range of industry contacts.

We are currently searching for individuals with a record of accomplishment in the following disciplines:

- **MICRO ARCHITECTURE**
- **COMPILER DESIGN**
- **OPERATING SYSTEMS DESIGN**
- **DATA BASE DESIGN**
- **NETWORKING**
- **DIGITAL LOGIC DESIGN**
- **FIRMWARE DESIGN**
- **ANALOG RF/MODEM DESIGN**

Clients include mainframe, mini, micro, and peripheral vendors; research organizations; software and systems houses; and consulting firms.

For a confidential discussion please write; or call Rande McCollum or Sally Silver at (617) 235-8505.

Winter, Wyman & Company
MEMBER
MASSACHUSETTS PROFESSIONAL
PLACEMENT CONSULTANTS

Wellesley Office/60 William Street Wellesley, Massachusetts 02181

Reader Service Number 12