INTRODUCTION

During the past several years, industry has seen an explosion in the cost of software production coupled with a decline in the quality and reliability of the results. A realization that structured programming, top-down design, and other changes in techniques can help has alerted the field to the importance of applying advanced design and programming methods to software production.1,2

For the past four years, Caine, Farber & Gordon, Inc. has used such advanced techniques as structured programming, top-down design and system implementation, centralized program production libraries, and egoless programming teams for all of its programming.3-5 With these techniques we have achieved a level of productivity comparable to that recently reported by others employing similar techniques.

However, within the last year, we greatly refined these techniques, applying them to design as well as to programming. This has resulted in increased productivity, greatly decreased debugging effort, and clearly superior products. On recent complex projects we have achieved production rates, over the full development cycle, of 60-65 lines of finished code per man-day and computer utilization of less than 0.25 CPU hours per thousand lines of finished code. For comparison, these production rates are approximately half again better than our best efforts using just structured programming techniques and 4-6 times better than average industrial experience using classical techniques. Computer usage was four times smaller than our experience with just structured programming techniques and more than 10 times smaller than classical industrial averages.

As an example, consider the two CFG projects shown in Table 1. Project “A” is a major component of a seismic data processing system for oil exploration. It was produced using “classical” structured programming techniques and production rates compare favorably to other projects which used similar techniques. Project “B” is a system for the automatic restructuring of Fortran programs. It was developed using the latest CFG methods. Production rates were 50 percent better than for project “A” and the amount of computer time used in development was approximately one quarter of that used for the first project. In each case, a “line” of code was taken to be one 80-column source card with common data definitions counted only once. Both projects were developed using an IBM 370/158.

In order to achieve the results that we are currently experiencing, we have developed a comprehensive software production methodology which places its greatest emphasis on design. Before any code is written, a complete design is produced which contains:

- all external and internal interface definitions
- definitions of all error situations
- identification of all procedures
- identification of all procedure calls
- definition of all global data
- definition of all control blocks
- specification of the processing algorithms of all procedures

The design is produced and presented top-down and is oriented toward understandability by people. While in no sense is our design process automated, it is supported by a series of tools—both computerized and procedural.

This paper is not intended to present our complete design and implementation methodology. Rather, it discusses one of the design tools—the “Program Design Language” (PDL) and its computerized processor. Both of these have been in extensive use since the autumn of 1973.

THE PURPOSE OF PDL

PDL is designed for the production of structured designs in a top-down manner. It is a “pidgin” language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language). In a sense, it can be thought of as “structured English.”

While the use of pidgin languages is also advocated by others, we have taken the additional steps of imposing a degree of formalism on the language and supplying a processor for it. Input to the processor consists of control information plus designs for procedures (called “segments” in PDL). The output is a working design document which can, if desired, be photo-reduced and included in a project development workbook.

The output of the processor completely replaces flowcharts since PDL designs are easier to produce, easier to change, and easier to read than are designs presented in flowchart form.
TABLE 1—Production Comparisons

<table>
<thead>
<tr>
<th></th>
<th>PROJECT &quot;A&quot;</th>
<th>PROJECT &quot;B&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVELOPMENT METHOD</td>
<td>CLASSICAL STRUCTURED</td>
<td>LATEST CFG</td>
</tr>
<tr>
<td>PROGRAMMING LANGUAGE</td>
<td>PL/I DIALECT</td>
<td>PL/I</td>
</tr>
<tr>
<td>SIZE OF PROGRAM (LINES)</td>
<td>32,000</td>
<td>27,000</td>
</tr>
<tr>
<td>SIZE OF TEAM</td>
<td>3-6</td>
<td>3-5</td>
</tr>
<tr>
<td>ELAPSED TIME (MONTHS)</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>LINES PER MAN-DAY</td>
<td>40</td>
<td>65</td>
</tr>
<tr>
<td>CPU HOURS PER 1000 LINES (IBM 370/158)</td>
<td>0.90</td>
<td>0.16</td>
</tr>
</tbody>
</table>

DESIGNING FOR PEOPLE IN PDL

Like a flowchart, and unlike a program, PDL can be written with whatever level of detail is appropriate to the problem at hand. A designer can start with a few pages giving the general structure of his system and finish, if necessary, with even more precision than would exist in the corresponding program.

In our experience, the purpose of a design is to communicate the designer’s idea to other people—not to a computer. Figure 1 shows a sample design “segment” for a simple exchange sort. Note that we are not attempting to illustrate efficient sorting methods. Rather, having decided to use this particular sorting method, we wish to present the algorithm in a way that it can be easily comprehended. Given that the “DO UNTIL” construct represents a loop whose completion test occurs at the end of the loop, the operation of the algorithm is apparent. It is clearly better, from the viewpoint of understandability, than either the flowchart of Figure 2 or the translation of the algorithm into PL/I as shown in Figure 3.

A virtue of PDL is that a rough outline of an entire problem solution can be quickly constructed. This level of design can be easily understood by people other than the designer. Thus, criticisms, suggestions, and modifications can be quickly incorporated into the design, possibly resulting in complete rewrites of major sections. When the design has stabilized at this level, more detail can be added in successive passes through the design with decisions at each point affecting smaller and smaller areas.

THE FORM OF A DESIGN IN PDL

A design produced in PDL consists of a number of “flow segments,” each corresponding roughly to a procedure in the final implementation. A sample of a high-level flow segment from a large design is shown in Figure 4. If a statement in a segment references another flow segment,
SORT:
PROCEDURE(TABLE);
DECLARE TABLE(*) FIXED BIN;
DECLARE INTERCHANGED BIT(1);
DECLARE TEMP FIXED BIN;
IF DIM(TABLE,1) > 1 THEN
  DO;
  INTERCHANGED = '1'B;
  DO WHILE (INTERCHANGED);
    INTERCHANGED = '0'B;
    DO I=LBOUND(TABLE,1) TO HBOUND(TABLE,1)-1;
      IF TABLE(I)>TABLE(I+1) THEN
        DO;
          INTERCHANGED = '1'B;
          TEMP=TABLE(I);
          TABLE(I)=TABLE(I+1);
          TABLE(I+1)=TEMP;
        END;
      END;
    END;
  END;
END;
END;
END;
END SORT;

Figure 3—PL/I procedure for sorting algorithm

the page number of the referenced segment is shown to the left of the referencing statement. A sample low-level segment is shown in Figure 5.

The statements which compose a flow segment are entered in free form. The PDL processor automatically underlines keywords, indents statements to correspond to structure nesting levels, and provides automatic continuation from line to line.

Design information may also be entered in “text segments.” These contain purely textual information such as commentary, data formats, assumptions, and constraints.

The document output by the PDL processor is in a form ready for photo-reduction and publication. It contains:

- a cover page giving the design title, data, and processor identification
- a table of contents (Figure 6)
- the body of the design, consisting of flow segments and text segments
- a “reference tree” showing how segment references are nested (Figure 7)
- a cross-reference listing showing the page and line number at which each segment is referenced (Figure 8)

DESIGN CONSTRUCTS

What goes into a design segment is generally at the discretion of the designer. In choosing the form of presentation, he is guided by a compendium of style which has been developed through extensive experience. However, the language and the processor have been defined to encourage and support design constructs which relate directly to the constructs of structured coding. The two primary constructs are the IF and the DO.

The IF construct

The IF construct provides the means for indicating conditional execution. It corresponds to the classical IF...THEN...ELSE construct of Algol-60 and PL/I,
augmented by the ELSEIF of languages such as Algol-68. The latter is used to prevent excessive indentation levels when cascaded tests are used.

The general form of the construct is shown in Figure 9. Any number (including zero) ELSEIF's are allowed and at most one ELSE is allowed.

**The DO construct**

This construct is used to indicate repeated execution and for case selection. The reasons for the dual use of this construct are historic in nature and closely map several of the in-house implementation languages we frequently use. It may be effectively argued that a separate construct for case selection would be better.

The iterative DO is indicated by:

```
DO iteration criteria
one or more statements
ENDDO
```

The "iteration criteria" can be chosen to suit the problem. As always, bias toward human understandability is preferred. Statements such as:

```
DO WHILE THERE ARE INPUT RECORDS
DO UNTIL "END" STATEMENT HAS BEEN PROCESSED
DO FOR EACH ITEM IN THE LIST EXCEPT
THE LAST ONE
```

occur frequently in actual designs.

Our experience, and that of others, has shown that a provision for premature exit from a loop and premature repetition of a loop are frequently useful. To accomplish this, we take the statement

```
UNDO
```

to mean that control is to pass to the point following the ENDDO of the loop. Likewise,

```
CYCLE
```

is taken to mean that control is to pass to the loop termination test.

Since we may wish that an UNDO or CYCLE apply to an outer loop in a nest of loops, any DO may be labelled and the label may be placed after the UNDO or CYCLE. Case selection is indicated by

```
DO CASE selection criteria
```

Again, we advocate the use of understandable selection cri-
Generally, we use labels in the body of the DO to indicate where control passes for each case. This is illustrated in Figure 10.

**FUTURE DIRECTIONS**

The results we have achieved with PDL have exceeded our original expectations. However, it is clear that further development is both possible and desirable. The areas which we are currently exploring include:

- **handling of data:** The current PDL presents a procedural design—a design of control flow and processing actions. It would be very desirable to have a similar mechanism for the design of data structures and data flow. A method for integrating the data and procedural designs and performing mutual cross-referencing would be very powerful, indeed.

- **interactive versions:** the current PDL processor is batch oriented. The ability to compose and, more importantly, to modify a design on-line in a manner specifically planned for interactive use would be of great assistance. This would be particularly advantageous during the early stages of a project when design changes are often frequent and extensive.

- **total design system:** an integrated computer system for software design, such as the DBS system of Professor R. M. Graham, is a natural outgrowth of our work with PDL. Such a system would act as an information management system maintaining a data base of designs. Designs could be entered and modified; questions about a design and the inter-relations of its parts could be asked and answered; reports on design status and completeness could be prepared. Provision for simulation of a design for performance estimation and a mechanism for transition from design to code are also important.

**CONCLUSIONS**

In the autumn of 1973, we integrated the use of PDL and its processor into our software design and implementation methodology. Since then, it has been used on a number of
projects of varying sizes. The results have been comparable to those discussed earlier.

PDL is not a "panacea" and it is certainly possible to produce bad designs using it. However, we have found that our designers and programmers quickly learn to use PDL effectively. Its emphasis on designing for people provides a high degree of confidence in the correctness of the design. In our experience, it is almost impossible to "wave your hands" in PDL. If a designer doesn’t really yet see how to solve a particular problem, he can't just gloss over it without the resulting design gap being readily apparent to a reader of the design. This, plus the basic readability of a PDL design, means that clients, management, and team members can both understand the proposed solution and gauge its degree of completeness.

We have also found that PDL works equally well for large and small projects. Because it is so easy to use, persons starting to work on even a "quick and dirty" utility will first sketch out a solution in PDL. In the past, such programs were usually written with little or no design preceding the actual coding.

REFERENCES