Use of Functional Blocks in the Design, Coding, and Checkout of FORTRAN Modules

Ronald C. Wackwitz
COMP-AID
513 Old Bear Creek Road
New Braunfels, TX 78132

October 02, 1989
Revised August 09, 2004

Contents

List of Tables

List of Figures

Use of Functional Blocks in the Design, Coding, and Checkout of FORTRAN Modules

Ronald C. Wackwitz
COMP-AID
513 Old Bear Creek Road
New Braunfels, TX 78132

October 02, 1989
Revised August 09, 2004

Abstract

The use of the COMP-AID Company's *Functional Block Outline* methodology to design and implement the algorithmic representation of a new module, when used in conjunction with the **RENUMF**© program, results in a coding rate of 12.9 Internal Statement Numbers (**ISN**s) per hour of completely checked out and documented high-quality code, which is from 2 to 5 times greater than comparably available values reported by Boehm (1981).

This value is based on carefully timed studies conducted on 12 modules — as displayed in Figure 1 on page 9. The size of a module, in effective ISNs, *excludes* JCL, comment lines, and variable declarations. Moreover, each ISN (which represents a single FORTRAN statement) counts as *one* ISN, whether the statement occupies only a single line or 20 lines.

1 Introduction — Can We Have *Productivity* and *Quality* Simultaneously?

Is greater programmer productivity the *primary* need of all FORTRAN programming shops facing deadlines? While increased productivity seems an obvious enough requirement, yet Peter Craig, in an issue of *InformationWEEK*,[1] cautioned that this is not enough:

> "In discussing possible cure-alls for the software development backlogs plaguing MIS shops, those who insist on improved programmer productivity as the obvious solution to the problem are not on target."

After stating that the solution doesn't lie in overt assaults on lines of code per man-year, he went on to emphasize that,

> "Rather, the solution lies in recognizing and dealing with MIS's fundamental software-development problem, which involves not programmer productivity, but the quality of the end product."

[1] Peter M. Craig, Program Quality Eclipses Programmer Productivity. *InformationWEEK*, November 23, 1987, page 31.

In the long-run of things, I'm sure we'd all agree with this. But in the short-run, when that current deadline's upon us, I'm afraid many of us will go with getting the code out — quality aside. Moreover, we might rationalize that quality is a somewhat nebulous term,[2] meaning different things to different people. And when I'm under a deadline, I may just view quality code as "timely" code.

This article purports to show that the use of the COMP-AID *Functional Block Outline* (**FBO**) methodology to design and implement the algorithmic representation of a new module, when used in conjunction with the **RENUMF**© program, not only results in a coding rate which is from 2 to 5 times greater than comparably available values reported by Boehm (1981),[3] but also effects highly understandable, completely documented code. This average coding rate of 12.9 ISNs per hour includes the time:

- to design/implement the module algorithm, by using the FBO methodology,
- to code the module, by filling in the completed FBO,
- to checkout the module (includes time to write the test driver),
- to complete internal documentation, including local variables.

This value is based on carefully timed studies conducted on 12 modules coded by the author — as displayed in Figure 1 on page 9, ranging from 68 ISNs (requiring 3.22 hours) to 397 ISNs (requiring 25.15 hours),while contracting through the COMP-AID Company for Texaco, in Houston, Texas, from 1987 to 1990. Six of these were the "spooling" modules[4] done for Bill Deuschle; four were UJS modules[5] done while under Phyllis Everett; and the last two were done in connection with working on the :OUTPUT processor under Gary Turpin.

An illustration of the use of the FBO methodology in the design/implementation of the algorithmic representation of the DOCALL and POSTN modules is presented in the following section. Sections 3 and 4 then describe *what* we are actually measuring, while Section 5 compares our measured coding rates with those of Boehm (1981). Section 6 describes *how* the methodology accomplishes its purpose.

2 An Illustration of the Use of the FBO Methodology

2.1 A Brief Summary of the Functional Block Outline Methodology

Since Section II of the COMP-AID document, "Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)" (Wackwitz and Falck, 1980), as well as our recent report,"Overview of The Methodology of Functional Blocks", adequately describe the concept of the Functional Block Outline, we shall here present only a briefest summary of it.

[2]Please see Table 5 on page 13 for a review of the important characteristics of *quality* code.

[3]These values are derived in Section 5 on page 12.

[4]These modules were described by Bill Deuschle in a 09/30/87 Texaco internal document entitled "Network Services Spool Management Subsystem Design Document".

[5]The author designed and coded many modules while he served as a member of the UJS team, first under Steve Fowlkes, and subsequently under Phyllis Everett. Unfortunately, only four of these modules were accurately timed.

Just as an entire *program* can be decomposed into a set of independent functional modules, so also then can any individual *module* be decomposed into a set of independent functional blocks, where these blocks of code constitute the steps required to accomplish the module task.

These individual *Functional Blocks* within a module are very similar — from a logical standpoint — to an individual subroutine or function within a program. For example, for a programmer to use a single entry subroutine such as **SORT (A, II, JJ)**, which sorts the **A** array into increasing order from A(II) to A(JJ), he knows that he must first load the **A** array and define **II** and **JJ** before calling SORT. That is, defining these three arguments (A, II, JJ) meets the entry requirements expected by SORT. In other words, it is the logical-state-of-affairs that must be true upon entry into SORT for the SORT subroutine to work correctly.

We can similarly look at each functional block of code within a given module, since each block expects certain variables to be defined, or certain events to have occurred, prior to its being invoked. Looked at another way, if execution starts at a block of code, then a certain logical-state-of-affairs must exist. Moreover, this logical-state-of-affairs required for entry into that block should be *unique*, independently of how control arrives there.[6]

Since each block of code within a module performs a desired function only when a required logical-state-of-affairs is true, we can enhance our understanding of the algorithmic representation of the module by preceeding each block of code by the single line of comment in the form,

C <*Event*>: *Process-description*,

where *event* refers to the logical-state-of-affairs that must be true upon entry (i.e., the logical entry assertion), and *process-description* refers to the process to be performed.

Because of the complex interactions in a non-trivial module, it is virtually impossible to understand it *if we view it as a single entity*. However, if we can hierarchially decompose the module into a set of Functional Blocks, in which each Functional Block is *small enough* that we can readily see that it correctly performs its required function if its entry logical-assertion requirements (<*event*> or interface conditions) are correctly met, then we can use these small understandable Functional Blocks as building-blocks to construct increasingly complex, higher level Functional Blocks. Prof. Donald E. Knuth (Knuth, 1974, p. 291) succinctly summarizes this as,

> "We understand complex things by systematically breaking them into successively simpler parts and understanding how the parts fit together locally."

By restricting our use of the "<*Event*>: *Process-description*"s such that

[6]The undated COMP-AID document, "A Discussion of Non-Unique <*Event*>'s" touches upon this with respect to the **GO TO** syntax. Of course, with the **IF-THEN-ELSE** syntax, this condition is always assured. (Interestingly, the **IF-THEN-ELSE** syntax, while solving this problem, unfortunately opens up another entire set of problems, when too deeply nested, that can also adversely affect module maintainability.)

- the logical entry assertion, denoted by <*Event*>, is *unique* for each given block, regardless of how control reaches that block,

- entry to a target block from an external block is to the *beginning* of the target block, with the additional understanding that the target block be at the same — or higher — level as the originating block,

we transform them into a high-level design language,[7] which is exactly what we have been denoting by the **F**unctional **B**lock **O**utline.

Therefore, the FBO methodology requires that any new module be completely described[8] by its FBO *before* any coding is done. In fact, we hope to demonstrate that use of the FBO permits one to design and implement the algorithmic representation of the module algorithm significantly faster than attempting to do it by other means, such as jumping into the coding directly (the worst of all), or even the *direct* use of high-level pseudo code.[9]

2.2 Use of the DOCALL Program to Illustrate the FBO Methodology

It was of interest, while working in the :OUTPUT group under Gary Turpin, to know what modules called each given module within the execute phase. That is, starting with the root PROGRAM module, it was desired to generate an alphabetically sorted list of each associated module, with all the modules which called it listed across from it. It was decided to utilize step-wise refinement to accomplish this task, breaking the task into three phases:

- Generate an initial list of the modules, starting with the selected root module and listed in order or occurrence, with all the modules which each calls listed across from it. By "order of occurrence" we mean that each module encountered in the root module is then in turn searched for modules which it calls, and these in turn are then searched for each module which they call, until the search for all encountered modules is either exhausted or not found in the specified directory.
- Sort the above list of encountered modules into alphabetical order, with the modules that each of these modules calls listed across from it.
- Rearrange the above sorted list, so that now each module listed in alphabetical order contains across from it those modules which call it.

Since it was decided for the present to restrict our search to called modules only, thereby eliminating the search for functional usage, we named this program **DOCALL**, as it **DO**cumented the **CALL**ed modules.

[7] For example, Wackwitz and Falck (1980, Appendix A) illustrate how a FBO can be transformed into an equivalent Warnier-Orr diagram (Orr, 1977, pp. 35–106), which is a recognized high-level design language.

[8] We qualify *completely* to denote the requirements of the module as we presently understand them. Moreover, we may also produce a FBO for a simplified version of the final module, and then use stepwise refinement to achieve the final form of the module once the simplified form is coded and tested.

[9] We shall discuss the relation of pseudo code to the FBO further in Section 4.3.

2.3 Logging the Development Effort

Table 1 on page 6 summarizes the various development efforts involved in producing the DOCALL module, up through the second phase.[10]

2.4 How Do We Estimate Time?

Before proceeding, please note that Table 1 shows that — although work was spread over six distinct days — only 22.78 actual hours was charged to the 358 effective ISNs, as opposed to 48 hours, assuming 8 hours/day. I did this since, if one is attempting to accurately assess one programming methodology over another, then one must obviously use the actual times.

However, if one is attempting to estimate the length of time that a proposed software project will take, using formulas based on prior experience, then one must realize that these formulas usually assume *minimal* interruption. Boehm (1981, p. 59), for example, cautions that the formulas presented in his book are predicated on the assumption that "the project will enjoy good management.... For example, the amount of nonproductive slack time is kept small by the manager and the customer."

Brooks (1975, p. 89) confirms this in the case of Charles Portman of the Computer Equipment Organization of Manchester, who found his teams missing schedules by almost one-half — despite the fact that the estimates were very carefully done by experienced teams estimating man-hours for several hundred subtasks on a PERT chart. Therefore, when the slippage pattern appeared, he asked them to keep careful daily logs of their time usage. These showed that his teams were only realizing 50% of the working week as actual development time. The remainder was accounted for by machine downtime, higher-priority short unrelated jobs, unrelated meetings and paperwork, company business, sickness, personal time, etc.

My own experience on the Spooling project adequately confirms this, as noted in Table 2, which shows that an average of only 69.5% of each working day was spent on the actual spooling project. The remainder was spent in required UJS-related work — namely meetings.

2.5 The First Three Passes at the FBO for the DOCALL Module

Since we had a nice Shell Sort algorithm at our disposal (which would help us in Phase 2), it was envisioned then that most of the effort would be spent on constructing the FBO for the first of the three phases.

First, Table 7 on page 17 shows the status of the FBO after 0.65 minutes, before taking a break. Sometimes — though not often — I enter coding statements as I go along, as seen in Block #1.1.2, when the process-description for that block is not sufficiently specific. Also, I always enter the declarations at the top as I proceed.

After 1.67 hours, the first two Level 1 Functional Blocks were complete, as seen from Table 8

[10] It was intended to separate the development effort for the second phase into a separate summary table, as we did do for the third phase (cf. Table 4 on page 9). Unfortunately, I had completed the second phase before I realized that I hadn't documented the module (the GTD header and the local variable) for the first phase. Therefore, I just lumped the first two phases together.

Table 1: Log of Initial Coding Effort on **DOCALL** Module

Major Function	Date	Time	Elapsed time, hrs	Comments
FBO	08/25/89	10:54		
	"	11:33	0.65	Completed FBO #1 — cf. Table 7
	"	12:45		
	"	13:00	0.25	
	"	13:26		
	"	14:12	0.77	Completed FBO #2 — cf. Table 8
	"	14:12		
	"	14:49	0.62	
	"	15:17		
	"	16:28	1.18	Completed FBO #3 — cf. Tables 9 through 10
	08/26/89	18:11		Starting looking at GETJCL to help on parsing
	"	18:23	0.2	algorithm
	"	18:39		Attempted to expand Block #s 2.4 & 2.5 - realized
	"	19:35	0.93	needed to put this logic into a separate module; thus broke off for work on the new POSTN module.
	08/27/89	15:08		Revamped Block #2, utilizing the POSTN
	"	16:00	0.87	function.
	08/28/89	11:19		Incorported Block #2 into the FBO of DOCALL —
	"	11:36	0.28	cf. Table 12 for Updated Block #2
	"	13:21		Added option to output to Printer or File,
	"	13:50	0.48	as well as to Screen.
Coding	"	17:57		
	"	18:05	0.13	
	"	18:23		
	"	19:04	0.68	
	"	19:56		
	"	22:48	2.87	Down to start of Block #3.
	08/29/89	06:47		
	"	07:55	1.13	Resumed coding - got down to Block #4.5.4.
	"	09:00		
	"	09:20	0.33	Completed coding.
	"	09:20		Got to compile without errors; then ran RENUMF
	"	09:33	0.22	on it.
	"	10:38		
	"	11:19	0.68	Before testing, added the SUFFIX logic.
Testing	"	12:42		Coded 4 test modules, MAIN, MOD1, MOD2,
	"	12:59	0.28	& MOD3.
	"	13:22		
	"	16:11	2.82	In debug mode.
	"	16:49		
	"	18:15	1.43	Got to basically work.
	"	18:27		Completed testing based on specs of original
	"	18:52	0.42	FBO + SUFFIX.
Add-on #1:	"	20:39		Added a new Block #5 to produce a sorted
FBO + Coding +	"	21:27	0.8	output.
Testing	"	21:27		
	"	21:38	0.18	Tested this.
	"	21:38		
	"	22:22	0.73	Prettied up the output.
Documentation	08/30/89	09:04		
	"	09:58	0.90	
	"	10:13		
	"	11:21	1.13	
	"	12:49		
	"	13:15	0.43	Completed documentation of local variables.
	"	14:13		
	"	14:47	0.57	
	"	15:18		
	"	16:07	0.82	Completed header.

Major Function	Elapsed Time, Hours	%	No. ISNs
FBO	0.65 + 0.25 + 0.77 + 0.62 + 1.18 + 0.20 + 0.93 + 0.87 + 0.28 + 0.48 = 6.23	27	No. ISNs: 376
Coding	0.13 + 0.68 + 2.87 + 1.13 + 0.33 + 0.22 + 0.68 = 6.04	26	No. Declarations: 18
Testing	0.28 + 2.82 + 1.43 + 0.42 = 4.95	22	No. effective ISNs: 358
Add-on #1	0.80 + 0.18 + 0.73 = 1.71	8	
Documentation	0.90 + 1.13 + 0.43 + 0.57 + 0.82 = 3.85	17	
Total	22.78	100	358

Table 2: Log of *Total* and *Spooling-Related* Hours Spent on SPOOLING Project

Comment	Day	Date	Hours, Total	Hours, Proj.	% Proj.
	Wed	10/07/87	13.96	12.03	86.2
	Thurs	10/08/87	13.46	11.03	81.9
	Fri	10/09/87	6.37	5.19	81.5
	Mon	10/12/87	11.18	7.24	64.8
	Tues	10/13/87	13.23	9.04	68.3
	Wed	10/14/87	16.07	13.78	85.7
	Thurs	10/15/87	11.59	10.61	91.5
	Fri	10/16/87	9.21	7.61	82.6
	Mon	10/19/87	8.75	7.74	88.5
	Tues	10/20/87	12.55	7.89	62.9
	Wed	10/21/87	14.73	5.03	34.1
	Thurs	10/22/87	10.57	3.11	29.4
	Fri	10/23/87	8.32	6.2	74.5
	Mon	10/26/87	15.64	10.93	70.0
	Tues	10/27/87	12.42	5.1	40.8
Totals		15 days	178.05	122.53	1042.7
Avearge			11.87	8.17	69.5

on page 18, with the understanding, however, that Block #s 2.4 and 2.5 needed to be amplified on — that is, further clarified by being decomposed into Level 3 Functional Blocks.

And, by the end of 08/25/89, after only 3.47 hours, we had a semi-final FBO (as shown in Tables 9 and 10) which was complete except — as noted above — for the need to amplify on Block #s 2.4 and 2.5.

2.6 Reflecting on the Use of the FBO to Iteratively Design the Module Algorithm

The use of the Functional Block Outline methodology allows one to both *design* the algorithmic representation of the module and to *implement* it into the FORTRAN language. The process is iterative, following Wirth's (1971) suggestion, in which design is envisioned as a series of *refinement steps*. The *outline* style or mode of the FBO allows one to follow Wirth's advice to use as high a level of notation as possible.

As one designs/implements a module algorithm in the FBO, one is constantly adding blocks, deleting blocks, moving blocks — and thus renumbering the blocks in the process — as the algorithmic representation of the desired module function begins to take shape.[11] (That, by the way, is why it is crucial that all work with the FBO be done at the terminal, with little or no design of the FBO being done on paper.) For example, note the addition of the new Block #2.1[12] in Table 8, which was missing in Table 7. Also, note that the "→#X"[13] at the end of Block #2.2 in Table 8 has been replaced by "→#3" in Table 10, once the appropriate target block was identified during the iterative process.

2.7 Taking Time-Out to Code the POSTN Module

When I attempted to decompose Block #s 2.4 and 2.5 in the FBO shown in Table 10, I discovered that the logic required more than nine Level 3 blocks to decompose Block #2.4;

[11] Presently, all this is quite readily done with a source program editor (e.g., EDT on the VAX or PC-Write© on the PC).
[12] Interestingly, this step was later removed as unneeded.
[13] This denotes, "**GO TO** Block #X".

and that is always the "flag" in the FBO methodology which says to separate out current logic into — for example — a separate module. Thus I took time out — as noted in the entries for "08/26/89" in Table 1 — to design/code/test/document the new POSTN function, whose purpose was to find the position of a target string in a source string.

Table 3 below summarizes the various development efforts involved in producing the POSTN module, while Table 11 on page 21 shows its completed FBO.

Table 3: Log of Coding Effort on **POSTN** Module

Major Function	Date	Time	Elapsed time, hrs	Comments
FBO	08/26/89	19:35		
	"	20:11	0.60	
	"	20:29		
	"	21:43	1.23	
	"	21:51		Stopped to talk to Maynell, who called about
	"	21:57	0.1	modem problems.
	"	23:13		
	"	23:18	0.08	Complete FBO
	"	23:18		
	"	23:24	0.10	Desk checkout of FBO
Coding	"	23:24		
	08/27/89	00:24	1.0	
Testing	"	13:21		
	"	13:41	0.33	Coded test program, TSTPOSTN
	"	13:41		
	"	14:00	0.32	Testing completed
Documentation	"	14:00		
	"	15:08	1.13	

Major Function	Elapsed Time, Hours	%	No. ISNs
FBO	0.60 + 1.23 + 0.10 + 0.08 + 0.10 = 2.11	43	No. ISNs: 59
Coding	1.00	21	No. Declarations: 5
Testing	0.33 + 0.32 = 0.65	13	No. effective ISNs: 54
Documentation	1.13	23	
Total	4.89	100	54

2.8 Revamping Block #2 of the DOCALL Module

With the checked out POSTN module now available, Block #2 — as shown in Table 10 — could now be revamped, as shown in Table 12 on page 22.

2.9 The Final Phase 1 FBO for the DOCALL Module

Therefore, the final FBO for Phase 1 coding of the DOCALL module is shown in Tables 13 through 16, starting on page 23. All this required 6.23 hours of effort, and occurred *before* a single line of coding was done. However, note that the POSTN module — which was observed to be necessary to DOCALL — was made available during the FBO process for DOCALL, but without any of its development effort being charged to DOCALL.

2.10 Completion of the FBO for Phase 3 of DOCALL, and a Plot of the Results

Finally, the log of the effort to add the Phase 3 capability to DOCALL through the addition of Block #6 is given in Table 4 below.

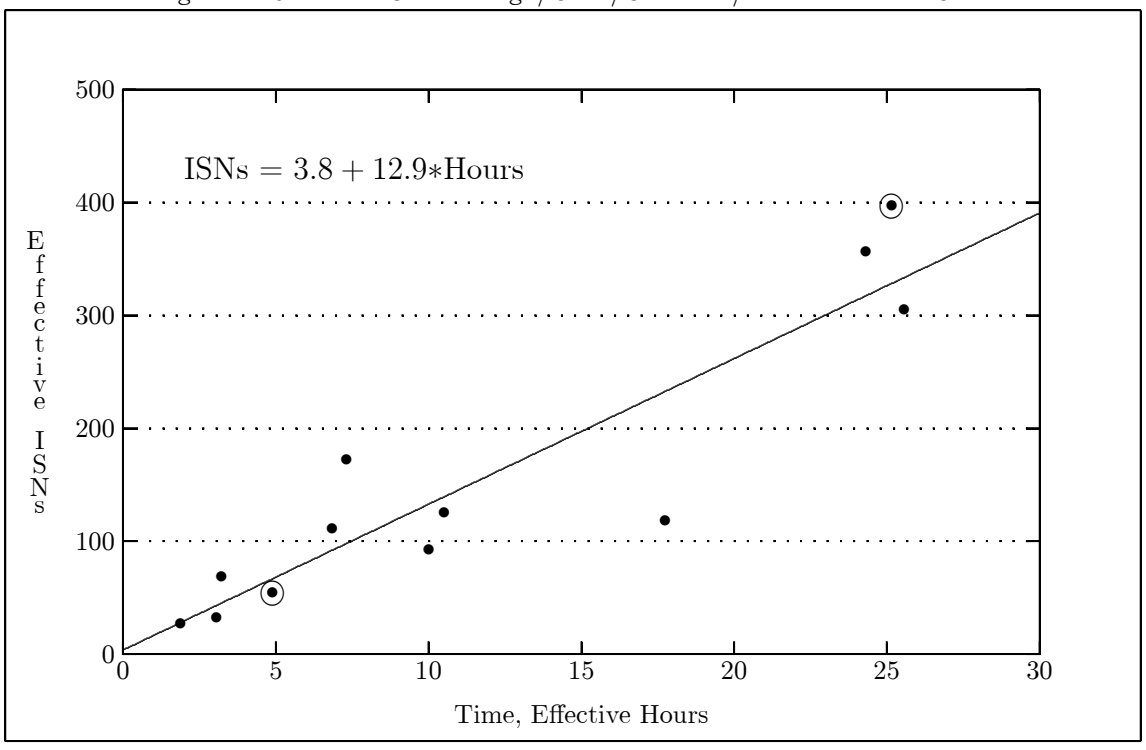
Table 4: Log of Secondary Coding Effort on **DOCALL** Module — Add on New Block #6

Major Function	Date	Time	Elapsed time, hrs	Comments
FBO	08/30/89	20:29		
	"	21:18	0.82	
Coding	"	21:18		
	"	22:23	1.08	
Testing	08/31/89	06:39		Even includes time to read Brenda's Thursday
	"	07:07	0.47	mail reminder

Major Function	Elapsed Time, Hours	%	No. ISNs
FBO	0.82	35	No. ISNs: 415
Coding	1.08	45	No. Declarations: 18
Testing	0.47	20	No. effective ISNs: 397
Total	2.37 + 22.78 = 25.15	100	397

Therefore, with the final ISN-*vs*-hours value for DOCALL of (397,25.15), along with that of (54,4.89) for the POSTN module from Table 11, we can plot these two points, along with the six spooling-project points and the four UJS-project points, in Figure 1 below, in which these two new points are enclosed in large circles. The linear least-squares curve fitted to the data does not imply linearity, since many authors (e.g., Brooks, 1975, pp. 88–91; Boehm, 1981, p. 57) have shown that the curve becomes exponential for very large systems. However, for small numbers of lines of code, the curve may well be approximated by a linear fit, which simply permits us to compare the productivities of different methodologies in this region.[14]

Figure 1: Use of FBOs to Design/Code/Checkout/Document New Code



[14]Also, we must keep in mind that Boehm's values apply to the cumulative count of *all* modules which constitute an entire project, whereas we are concerned only with values applicable to the *individual* modules within the project. Therefore, for individual modules, the representation shown in Figure 1 should always be in the linear region.

3 The Validity of our ISN or of Boehm's DSI

3.1 First, What Are We Attempting to Measure with the *ISN*?

We use the **I**nternal **S**tatement **N**umber (**ISN**), which represents a *single* FORTRAN statement (independently of whether it occupies one or 20 lines of code), in place of the usual *line of code*, or **D**elivered **S**ource **I**nstruction (**DSI**) as referred to by Boehm (1981, p. 57). Boehm's DSI refers to a line of code in just the *opposite* sense as does the ISN. That is, a single coding statement occupying five lines of code would count as five DSIs. (Interestingly, if a single line of code contained five coding statements, Boehm would still count it as a single DSI. This situation, however, should not be used in maintainable FORTRAN code.)

I feel that restricting a statement to count as a single DSI, even though it may occupy upto 20 lines of code, is more in keeping with relating each statement to a functional unit of thought, which is what takes the time in program design. While this is not true in the time required to code a very long statement, yet coding is so mechanical — once one knows what is to be coded — that I feel there is very small percentage in time difference involved in coding, for example, a one line statement and a five line statement.

At any rate, the use of the ISN as the counting device makes our figures somewhat more conservative than those of Boehm's, against whose coding rates we shall soon compare out values (cf. Section 5).

3.2 What Do We and Boehm Each Include in our "Lines of Code"?

The total DSIs for a delivered project, as defined by Boehm (1981, p. 59),

- include job control language (JCL) and DATA declarations,
- exclude comment cards, and unmodified utility software.

Our total ISNs for any single delivered module, on the other hand, exclude not only what Boehm excludes but also *exclude* the two items which Boehm includes — namely the JCL and the DATA declarations.[15]

Thus again, our total ISNs per module are somewhat conservative to those which Boehm gives.

3.3 Just How Valid Is the Line of Code in Measuring Productivity?

Boehm (1981, pp. 479–481) feels that the insensitivity with which the DSI is held as a unit of measuring productivity can be suitably countered. Moreover, he feels that several other such parameters which have been tried as a replacement (and he cites authors such as Halstead, McCabe, Curtis, etc.) run into even more difficult sensitivity problems than does the DSI

[15] I can see a rationale, however, behind including the DATA declarations; and in the future I shall do this, since they really replace assignment statements.

parameter. Obviously, one does not compares DSI values from assembly coding with those from FORTRAN coding. Also, he notes (1981, pp. 657–658) that one does not compare results of standard coding against values produced from a program generator.

4 What Does the Development Time Include?

4.1 A Comparison of Items Included in This and Boehm's Development Time

We have already stated that our productivity measurements include the time:

- to design/implement the module algorithm, by using the FBO methodology,
- to code the module, by filling in the completed FBO,
- to checkout the module (includes time to write the test driver),
- to complete internal documentation, including local variables.

The development period covered by Boehm's **CO**nstructive **CO**st **MO**del (COCOMO) cost estimates (Boehm, 1981, p. 59) "begins at the beginning of the product design phase (after successful completion of a software requirements review), and ends at the end of the integration and test phase (successful completion of a software acceptance review)". From a study of Boehm's Figure 5-1 (1981, p. 60), we see that this time also includes the time to *update* and *complete* the users' manual — but does not count the time to produce the initial draft, which was done in the requirements phase.

Because the COCOMO pertains to entire software projects, it contains two design stages, the "Product Design Specifications" and the "Detailed Design Specifications". The former would be relevant only to entire projects, while the latter would be relevant to each module constituting the project.

4.2 Product Design Specification Encountered in Spooling Project

I encountered this former design element while working on the spooling project, when I wound up with 29.4 additional hours that I could not attribute individually to any one of the six spooling modules shown in Figure 1:

- initial meetings with Bill (2.62 hours),
- general overview of pseudo code with Bill (0.78 hours),
- design of record layout (2.33 hours),
- design/modification of commons (6.38 hours),
- design and checkout of BUILDQUE program (1.39 hours),
- get and arrange source listings (2.01 hours),
- help users (8.92 hours),
- enhancement to original design (1.38 hours),
- producing the ERROR.LIST document (0.58 hours),
- initial version of the TESTSPOOL program (0.46 hours),

- final review of all modules with Bill (0.38 hours).

Subtracting out the 8.92 hours involved with helping users (as it was not directly related to producing the modules), I divided the remaining 20.48 hours by six, and added this 3.41 hours to the time for each of the six spooling modules. That is, the values plotted in Figure 1 for these six modules is each 3.41 hours greater than their combined hours due to design of the FBO, coding, checkout, and documentation.

Thus I am attempting to lump in this "product design" contribution into the productivity rate I am trying to measure, when applicable. Interestingly, no such contribution occurred for the DOCALL "system", as noted from Tables 1 and 4, due to its limited scope.

4.3 A Parting Clarification of the "Requirements" Phase

Noting that Boehm does not include the software requirements specification in his development effort, we need to ascertain exactly what we mean by it. First of all, it represents the initial description of *what* is to be done.

In the case of the Spooling Management System producded by Bill Deuschle, this included the time for Bill to produce all the high level pseudo code describing exactly how each module was to perform. Because the pseudo code was at a very high level of abstraction, it contained no detail of how these functions were to be implemented in the FORTRAN language; that was left for the development phase.[16]

In the case of the User Job Services (UJS) project associated with Starpak, this included the time to prepare a user's manual which described every function to be implemented.

5 A Comparison with Boehm's Basic COnstructive COst MOdel

Boehm's (1981, pp. 57–71) basic **CO**nstructive **CO**st **MO**del (COCOMO), which is based on years of experience with various size software projects (many with the U.S. Air Force), states that the effort equation for a small-to-medium size product developed in a familiar in-house environment is:

$$\text{MM} = 2.4(\text{KDSI})^{1.05}$$

The quantity **KDSI** is DSI in thousands, while **MM** denotes estimated Man-Months, which the COCOMO equates to 152 hours of working time per month.

Using the cited value of 152 hours per working month, we can convert his equation to 0.381 hours/DSI, which is 2.6 DSI/hour. Thus our cited rate of 12.9 ISN/hour is **4.9** times greater than Boehm's predicted value — which is based on the experience of many actual software projects.

[16] If the high level pseudo code is successively decomposed into a "lower level pseudo code", then that lower level pseudo code approaches the FBO in functionality.

However, noting Boehm's footnote reference (1981, p. 64) to Brooks (1975, pp. 4–7), in which he cites Brooks' caution that "a software product requires about three times as much effort to complete as an equivalent-sized personal software program", we shall multiply Boehm's rate of 2.6 DSI/hour by 3 to get 7.9 DSI/hour for production software, which when compared against our 12.9 ISN/hour now gives only **1.6** times greater.

Of course, if you read Brooks' comparison between the personal program and the "programming product", you will observe that we have included essentially everything in our values of development time that Brooks cites — except for producing the manual. Even here, however, Boehm only included the *updating* of the manual in the development time covered by his COCOMO equation. Thus I feel that the higher end of the observed "2 to 5 times greater" range is applicable.

6 Why the FBO Permits *Rapid* Production of *Quality* Code

6.1 The FBO Emphasizes the Production of Quality Code

First, we need to define *quality code*. We summarize what we view as the important characteristics of quality code in Table 5 below.

Table 5: Important Characteristics of Quality Code

Characteristic	Comments
Separate functions identified	The separate functions which constitute the task performed by the module must be readily identifiable — i.e., the start of each function within the module should be readily discernable. This is an alternate definition of *clarity*, by which any new programmer can study the module and quickly grasp the various parts which permit the module to perform its function.
Top-down structure	Not only does this mean that there is an absence of convoluted code (which the misuse of the **GO TO** can cause); it also means that each of the functional parts that make up the module are *independent* of each other. That is, making a change in how the third functional block in the module, for example, performs its purpose will have no effect on the remaining blocks of the module. (Of course, since each of the functional parts of the module communicates across their interfaces (e.g., what variables a block expects the prior executing block to provide) then if the interface requirements change, that will obviously affect some of the other parts of the module.)
Portable	In the absence of a specific requirement for great execution speed, *portablity* should always be emphasized when coding a module. Moreover, experienced programmers have learned how to achieve portable code *without* sacrificing execution speed.
Design for modification	Part-and-parcel with writing portable code is writing code which easily accomodates change. The following (which are all covered in the GTD *Programming Techniques Manual*) include the use of PARAMETER constants for dimension size or for program constants, the use of the (*) and *(*) respectively for dimension size and character length for arguments within the called module, etc.
Consistent style or organization	The use of a consistent programming style (and this means within the code itself as well as within the GTD header) permits a new programmer to readily find his or her way around in the module. This is another aspect of *clarity*.

The requirements inherent in the FBO satisfy these characteristics of quality code by

- permitting rapid identification of the functional blocks of code,
- producing top-down code with no convoluted coding,
- effecting independence of the functional blocks from each other,
- achieving a consistent style or organization,
- achieving code amenable to modification (maintenance and enhancement).

The last attribute is very important, since Brooks (1975, p. 122) states,

> "The fundamental problem with program maintenance is that fixing a defect has a substantial (20–50 percent) chance of introducing another."

The testimony to the reliability of software produced by the FBO is impressive:[17]

- **UJS.** Thanks to the accurate design by Steve Fowlkes and Phyllis Everett, along with the use of the FBO methodology on *all* UJS software coded in FORTRAN, the UJS system proved to be perhaps the most reliable software system within the entire Starpak project. It had very few errors, was proven readily amenable to correction and enhancement, and was highly portable. (For example, *without* the use of the UNIX "CURSES" software, UJS software was portable to any terminal supporting the ASCII Escape sequences, requiring only the single system-dependent module which gets a single character from the key board in unbuffered mode. Thus it was transportable to *both* UNIX and non-UNIX systems.)
- **Spooling Modules.** Bill Deuschle testified[18] not only that the spooling software was proven remakably reliable, but also that it was easily transported from the VAX to the Cray and IBM.
- **:OUTPUT Processor.** After structuring all the modules associated with the :OUTPUT processor by the methodology of Functional Blocks, the instance of reported errors essentially vanished. While these modules were not originally coded using the FBO methodology, yet this demonstrated an example of the combined use of the FBO and RENUMF to "reclaim" difficult-to-understand modules, such that — quality-wise — they have all the attributes as if they had originally been coded in Functional Blocks.

6.2 The FBO Aids the Programmer in Designing/Implementing the Module Algorithm

The FBO methodology aids the programmer in the design and implementation of the algoritmic representation of the module in five ways:

1. It accomodates Wirth's (1971) method of viewing program construction as a sequence of *refinement steps*.
2. The use of the "<*Event*>: *Process-Description*" notation is at a sufficiently high level (i.e., sufficiently cryptic or abstract) as to permit the designer to rapidly brainstorm various algorithmic approaches with a minimum of rewriting.
3. Because each "<*Event*>: *Process-Description*" entered occupies only one line, the programmer always has a large global view of the design/implementation as it unfolds.

[17] Obviously, an initial good design is also critical, since the very best implementation of badly designed software will still result in software that does not adequately meet the users' needs. Of course, if the design is "just a little bad", then an initial implementation by the FBO methodology will "fairly readily" accomodate a transformation to the desired specifications.

[18] Internal Texaco memo, dated 09 November 1988, commending Mr. Wackwitz for his support on the Spooling project.

4. It permits the programmer to produce the FBO on his or her terminal, as opposed to writing the "<*Event*>: *Process-Description*"s on paper, which would severely impede the expression of the flow of ideas. (Of course, for very long FBOs, I shall often print out a copy of the current status of the FBO, so that I can still view that portion that is no longer showing on the screen, thereby maintaining my global view.
5. It forces the programmer to concentrate on the design.

The last reason is very crucial. Brooks (1975, p. 20), for example, emphasized that for some years he had been using the following rule of thumb for scheduling a software task:

- $\frac{1}{3}$ planning
- $\frac{1}{6}$ coding
- $\frac{1}{4}$ component test and early system test
- $\frac{1}{4}$ system test, all components in hand.

I point this out to emphasize the large portion (of 1/3) that he had set aside for planning, since his planning stage is equivalent to the the production of a FBO for a module prior to coding it.

For example, Table 6, which itemizes the hours spent[19] on each of the 12 modules plotted in Figure 1, shows a comparable overall average of 40.4% spent on the initial production of the FBO. Think of that — spending 40.4% of your effort to produce a module *before* a single line of coding is done! But the end results are impressive.

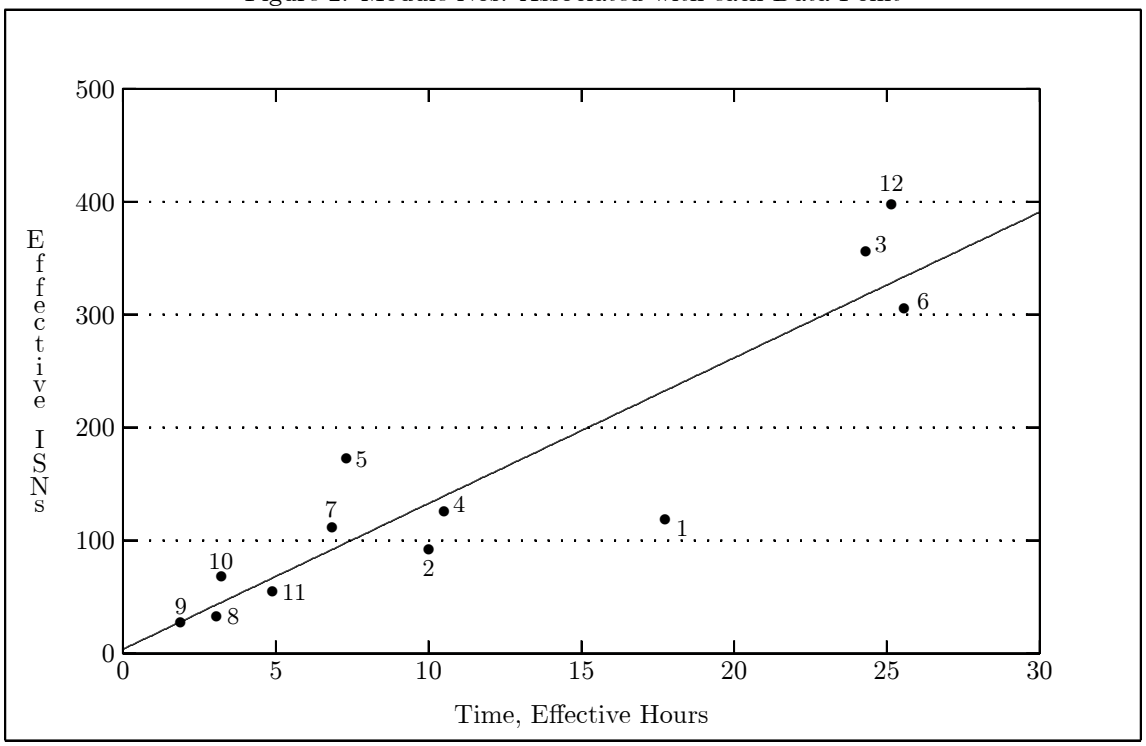
Table 6: Itemization of Time Spent on each of the 12 Modules Studied

Comment	Module No.	Module Name	FBO	Coding	Checkout	Documentation	% Time, FBO
	1	NSSPCR	6.2	3.47	3.47	1.19	43.3
	2	NSSPGC	2.21	2.1	1.87	0.42	33.5
	3	NSSPUD	11.41	4.57	2.94	1.98	54.6
	4	NSSPRD	2.62	2.13	1.2	1.15	36.9
	5	NSSPDL	1.27	0.88	0.87	0.88	32.6
	6	NSSPIN	10.17	4.81	4.9	2.27	45.9
	7	VALSP	2.38	1.72	1.22	1.52	34.8
	8	ADDVRN	0.6	0.85	0.58	1.23	18.4
	9	UNKNON	0.65	0.43	0.48	0.32	34.6
	10	CLEANUP	1.07	1.23	0.42	0.5	33.2
	11	POSTN	2.11	1.0	0.65	1.13	43.1
	12	DOCALL	6.23	6.04	4.95	3.85	29.6
Totals			46.92	29.23	23.55	16.44	440.5
Average			3.91	2.44	1.96	1.37	40.4

The "Module No." cited in this table are shown in Figure 2 for each module-point plotted. This is interesting, since it reveals that the point lying furtherest off the curve (Point #1) is that of the *first* spooling module coded. Obviously, more of the overall "Product Design Specification" discussed back in Section 4.2 is lumped into it — most noticeably in its FBO value.

[19]The 3.41 hours added to each of the six spooling modules is *not* included in these totals. Also, the values used for DOCALL are taken from Table 1, with the value of 1.71 hours due to the Phase 2 work excluded, as well as the 2.37 hours shown in Table 4 for the Phase 3 work excluded.

Figure 2: Module Nos. Associated with each Data Point



7 Conclusion

The **F**unctional **B**lock **O**utline (**FBO**) methodology developed by the COMP-AID Company permits *both* the achievement of high productivity rates for the design, coding, testing, and documentation of a module, and the achievement of high quality code (readily understandable, maintainable, and enhanceable). The rate of 12.9 ISN/hour at which this has been achieved is based on carefully timed studies of twelve modules, ranging from 68 ISNs (requiring 3.22 hours) to 397 ISNs (requiring 25.15 hours).

The productivity rates achieved are from 2 to 5 times as great as those reported by Boehm (1981), based on a study of a large data base of software projects, with the higher end of this range being favored.

A Tables of Functional Block Outlines

Table 7: Functional Block Outline #1

```
PARAMETER (MAXMOD=1000)
DIMENSION NMCALD(MAXMOD), MODNAM(MAXMOD)
CH*1  CTERM, CQUOTE, CAPSTR
CH*15 FNMAIN, MODNAM
CH*30 PTHNAM,
CH*60 FNAMFQ
CH*80 CARD

------------------------------------------------------------------------
1. <Entry>: Initialization
      1.1. <Entry>: Define program constants; init counters.
             1.1.1. <Entry>: Define program constants.
             1.1.2. <Constants defined>: Init counters.
                    NMMODS = 1
                    NMCALD(1:MAXMOD) = 0
      1.2. <Conts & counters initd>: Define LUNs; open LUNIN & LUNOUT.
      1.3. <LUNs defined>: Get name of directory in PTHNAM
      1.4. <PTHNAM defined>: Get name of main program module in FNMAIN.
      1.5. <FNMAIN defined>: Set FNAMFQ = PTHNAM // FNMAIN.
      1.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             1.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#1.7.
             1.6.2. <FNAMFQ opened to LUNMOD>: Set MODNAM(1)=FNMAIN; ->#2.
      1.7. <FNAMFQ doesn't exist.  Notify user; ->#1.3.

2. <FNAMFQ opened to LUNMOD>: Read thru, processing each "CALL " found.
      2.1. <Entry>: Read next line; if EOF on LUNMOD, ->#X.
      2.2. <Next line in CARD>: Upper-case it.
      2.3. <CARD upper-cased>: Scan for "CALL"; if none found, ->#2.1.
      2.4. <"CALL" found>: Scan for  MDNAME & "(" following; if none, ->#2.1.
      2.5. <Name in MDNAME>: Process it.
```

Table 8: Functional Block Outline #2

```
MAXMOD         I     MAX No. MODules permitted
MAXCLD         I     MAX No. of MODules that may be CaLleD by a given module
NMMCLD  MAXMOD I     No. of Modules CaLleD by this module
MODNAM  MAXMOD CH*15 Array of MODule NAMes encountered
MODCLD  MAXMOD,
        MAXCLD CH*15 Array of MODules CaLleD for ith module: MODNAM(i)
MNDXCR         I     INDeX into MODNAM array for FNAMCR
MINDEX         I     INDEX into MODNAM for called MDNAME
FNAMCR         CH*15 NAMe of CuRrent File (module) being processed
NMCMOD         I     NuMber of MODules currently called by FNAMCR
MDNAME         CH*15 NAME of called MoDule
-------------------------------------------------------------------------
PARAMETER (MAXMOD=1000)
DIMENSION NMMCLD(MAXMOD), MODNAM(MAXMOD)
CH*1  CTERM,  CQUOTE, CAPSTR
CH*15 FNAMCR, MODNAM, MDNAME, MODCLD
CH*30 PTHNAM,
CH*60 FNAMFQ
CH*80 CARD

-------------------------------------------------------------------------
1. <Entry>: Initialization
      1.1. <Entry>: Define program constants; init counters.
             1.1.1. <Entry>: Define program constants.
             1.1.2. <Constants defined>: Init counters.
                    NMMODS = 1
                    NMMCLD(1:MAXMOD) = 0
      1.2. <Conts & counters initd>: Define LUNs; open LUNIN & LUNOUT.
      1.3. <LUNs defined>: Get name of directory in PTHNAM
      1.4. <PTHNAM defined>: Get name of main program module in FNAMCR.
      1.5. <FNAMCR defined>: Set FNAMFQ = PTHNAM // FNAMCR.
      1.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             1.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#1.7.
             1.6.2. <FNAMFQ opened to LUNMOD>: Set MODNAM(1)=FNAMCR; ->#2.
      1.7. <FNAMFQ doesn't exist.  Notify user; ->#1.3.

2. <FNAMFQ opened to LUNMOD>: Read thru, processing each "CALL " found.
      2.1. <Entry>: Get index of FNAMCR in MODNAM array, MNDXCR.
      2.2. <MNDXCR defined>: Read next line; if EOF on LUNMOD, ->#X.
      2.3. <Next line in CARD>: Upper-case it.
      2.4. <CARD upper-cased>: Scan for "CALL"; if none found, ->#2.1.
      2.5. <"CALL" found>: Scan for = sign; if found, ->#2.1.
      2.6. <No = sign>: Scan for  MDNAME & "(" following; if none, ->#2.1.
      2.7. <Name in MDNAME>: Add to list of modules called by FNAMCR; ->#2.2.
             2.7.1. <Entry>: If MDNAME entered, skip ->#2.7.3.
             2.7.2. <Not yet entered>: Initialize it; drop.
                      2.7.2.1. <Entry>: Increment NMMODS.
                      2.7.2.2. <NMMODS inc>: Set MODNAM(NMMODS)=MDNAME; drop.
             2.7.3. <MDNAME in MODNAM>: Inc NMMCLD(MNDXCR); save in NMCMOD.
             2.7.4. <NMCMOD defined>: Set MODCLD(MNDXCR,NMCMOD)=MDNAME; ->#2.2.
```

Table 9: Functional Block Outline #3 — Part 1 of 2

```
MAXMOD         I     MAX No. MODules permitted
MAXCLD         I     MAX No. of MODules that may be CaLleD by a given module
NMMCLD  MAXMOD I     No. of Modules CaLleD by this module
MODNAM  MAXMOD CH*15 Array of MODule NAMes encountered
MODCLD  MAXMOD,
        MAXCLD CH*15 Array of MODules CaLleD for ith module: MODNAM(i)
MEXIST  MAXMOD L     Array of flags indicating Modules EXIST
NMCOL          I     NuMber of COLumn positions available for storing names
IACOL   10     I     Integer Array of starting COLumn positions
MNDXCR         I     INDeX into MODNAM array for FNAMCR
MINDEX         I     INDEX into MODNAM for called MDNAME
FNAMCR         CH*15 NAMe of CuRrent File (module) being processed
NMCMOD         I     NuMber of MODules currently called by FNAMCR
MDNAME         CH*15 NAME of called MoDule
NMMODS         I     NuMber of MODuleS currently encountered
NMMPRC         I     NuMber of Modules currently PRoCessed
MAXLNG         I     MAX LeNGth of the MODule names.
LENMOD         I     LENgth of longest module name
MAXLIN         I     MAX length of LINE used for printing
LINE           CH*132LINE used to hold each line of printed output
-------------------------------------------------------------------------
PARAMETER (MAXMOD=500, MAXCLD=100)
DIMENSION MODCLD(MAXMOD,MAXCLD)
DIMENSION NMMCLD(MAXMOD), MODNAM(MAXMOD), MEXIST(MAXMOD)
DIMENSION IACOL(10)
LOG    MEXIST
CH*1   CTERM,  CQUOTE, CAPSTR
CH*15  FNAMCR, MODNAM, MDNAME, MODCLD
CH*30  PTHNAM,
CH*60  FNAMFQ
CH*80  CARD
CH*132 LINE
-------------------------------------------------------------------------
1. <Entry>: Initialization
      1.1. <Entry>: Define program constants; init counters.
             1.1.1. <Entry>: Define program constants.
                    MAXLEN = LEN(MODNAM(1))
                    MAXLIN = 80
             1.1.2. <Constants defined>: Init counters.
                    NMMODS = 1
                    NMMPRC = 0
                    NMMCLD(1:MAXMOD) = 0
      1.2. <Conts & counters initd>: Define LUNs; open LUNIN & LUNOUT.
           LUNMOD = 1
           LUNIN  = 5
           LUNOUT = 6
      1.3. <LUNs defined>: Get name of directory in PTHNAM
      1.4. <PTHNAM defined>: Get name & length main module in (FNAMCR,LENMOD)
      1.5. <(FNAMCR,LENMOD) defined>: Set FNAMFQ = PTHNAM // FNAMCR.
      1.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             1.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#1.7.
             1.6.2. <FNAMFQ opened to LUNMOD>: Set MODNAM(1)=FNAMCR; ->#2.
      1.7. <FNAMFQ doesn't exist.  Notify user; ->#1.3.
```

Table 10: Functional Block Outline #3 — Part 2 of 2

```
2. <FNAMFQ opened to LUNMOD>: Read thru, processing each "CALL " found.
      2.1. <Entry>: Get index of FNAMCR in MODNAM array, MNDXCR.
      2.2. <MNDXCR defined>: Read next line; if EOF on LUNMOD, ->#3.
      2.3. <Next line in CARD>: Upper-case it.
      2.4. <CARD upper-cased>: Scan for "CALL"; if none found, ->#2.2.
      2.5. <"CALL" found>: Scan for = sign; if found, ->#2.2.
      2.6. <No = sign>: Scan for  MDNAME & "(" following; if none, ->#2.2.
      2.7. <Name in MDNAME>: If the same as FNAMCR, notify & ->#2.2.
      2.8. <MDNAME non-recursive>: Add to module list called by FNAMCR; ->#2.2.
             2.8.1. <Entry>: If MDNAME entered, skip ->#2.8.3.
             2.8.2. <Not yet entered>: Initialize it; drop.
                      2.8.2.1. <Entry>: Increment NMMODS.
                      2.8.2.2. <NMMODS inc>: Set MODNAM(NMMODS)=MDNAME; drop.
             2.8.3. <MDNAME in MODNAM>: Inc NMMCLD(MNDXCR); save in NMCMOD.
             2.8.4. <NMCMOD defined>: Set MODCLD(MNDXCR,NMCMOD)=MDNAME; ->#2.2.

3. <EOF on LUNMOD>: Close it; get next module in FNAMCR; open & ->#2.
      3.1. <Entry>: Close LUNMOD.
      3.2. <LUNMOD closed>: Increment NMMPRC.
      3.3. <NMMPRC inc>: If NMMODS=NMMPRC, all through; ->#4.
      3.4. <NMMPRC < NMMODS>: Set FNAMCR=MODNAM(NMMPRC+1); update LENMOD.
      3.5. <(FNAMCR,LENMOD) defined>: Set FNAMFQ = PTHNAM // FNAMCR.
      3.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             3.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#3.7.
             3.6.2. <FNAMFQ opened to LUNMOD>: ->#2.
      3.7. <FNAMFQ doesn't exist>: Process this case; ->#3.2.
             3.7.1. <Entry>: Notify user.
             3.7.2. <User notified>: Set MEXIST(NMMPRC+1)=False; ->#3.2.

4. <All modules scanned>: Now print out results.
      4.1. <Entry>: Compute NMCOL & IACOL for loading called module names.
             4.1.1. <Entry>: Set NMCOL = MAXLIN/(LENMOD+2) - 1
             4.1.2. <NMCOL defined>: Select each of NMCOL locs; EOLC=4.1.4.
             4.1.3. <I'th loc selected>: Set IACOL(I) = (LENMOD+2)*I
             4.1.4. <I'th loc processed>: Increment to next one.
      4.2. <(NMCOL,IACOL) defined>: Select each of the NMMODS modules; EOL=4.6.
      4.3. <I'th module name selected>: Branch on its MEXIST(I) flag.
      4.4. <I'th module doesn't exist>: Print out on single line; ->EOL.
      4.5. <I'th module exists>: Process this case; drop.
             4.5.1. <Entry>: Set NMCMOD=NMMCLD(I).
             4.5.2. <NMCMOD defined>: Select each of NMCMOD names; EOLC=4.5.6.
             4.5.3. <J'th name selected>: If MOD(J,NMCOL)<>1, ->#4.5.5.
             4.5.4. <At start of new line>: Init LINE, depending on J.
                      4.5.4.1. <Entry>: If J=1, ->#4.5.4.3.
                      4.5.4.2. <J<>1>: Print existing LINE; blank LINE; ->#4.5.5
                      4.5.4.3. <J=1>: Blank LINE; drop.
                      4.5.4.4. <Blank>: Set LINE(1:LENMOD)=MODNAM(I); drop.
             4.5.5. <Start of LINE defined>: Set LINE(IACOL(J): )=MODCLD(I,J)
             4.5.6. <J'th name inserted>: Increment to next one.
      4.6. <I'th module printed>: Increment to next module.

5. <All through>: Terminate.
```

Table 11: Functional Block Outline for the POSTN Module

```
INTEGER FUNCTION POSTN (NTARGT, CTARGT, RGDBLK, RGDAPS, NSOURC, CSOURC,
     *                  ISCOL,  CTERM)

------------------------------------------------------------------------
INTEGER
LOGICAL RGDBLK, RGDAPS
CHARACTER*(*) CTARGT, CSOURC, CTERM
------------------------------------------------------------------------
CHARACTER*1  CS, CT, CAPSTR, CFIRST
CHARACTER*80 CLINE
------------------------------------------------------------------------

1. <Entry>: perform initialization.
      1.1. <Entry>: Init functions parms: Zero POSTN.
      1.2. <POSTN=0>: Init (NS,APSTIN): NS=ISCOL & APSTIN=False.
      1.3. <Scan parms initd>: Set CFIRST=CTARGT(1:1).

2. <Target string not yet found>: Attempt to locate 1st target char.
      2.1. <Entry>: If (NS+1)>NSOURC, return
      2.2. <(NS+1)<=NSOURC>: Get next source char in UC in (CS,IC); inc NS.
      2.3. <UC char in CS & IC>: If Blank & RGDBLK=False, ->#2.1.
      2.4. <Not valid blank>: If not in apostrophe mode, ->#2.6.
      2.5. <APSTIN=T>: If "'", set APSTIN=F.   ->#2.1.
      2.6. <APSTIN=F>: If not "'" or RGDAPS=F, drop; else set APSTIN=T & ->#2.1
      2.7. <Not "'">: If TAB, ->#2.1.
      2.8. <Not TAB>: If CTERM, return.
      2.9. <Not CTERM>: If not CFIRST, ->#2.1; else init for scan and drop.
             2.9.1. <Entry>: If CS <> CFIRST, ->#2.1.
             2.9.2. <CS=CFIRST>: Save col-pos of CFIRST in CSOURC in NPOS.
             2.9.3. <NPOS defined>: If NTARGT>1, set NT=1 & ->#3.
             2.9.4. <NTARGT=1: Success>: Set POSTN=NPOS; update ISOURC; return.

   3. <NT Target chars agree so far>: See if next one does.
      3.1. <Entry>: Get next target char, CTARGT(NT+1:NT+1) in CT.
      3.2. <Next CT defined>: If (NS+1)>NSOURC, return.
      3.3. <(NS+1)<=NSOURC>: Get next source char in UC in (CS,IC); inc NS.
      3.4. <UC char in CS & IC>: If Blank & RGDBLK=False, ->#3.2.
      3.5. <Not valid blank>: If "'" & RGDAPS=T, ->#4.
      3.6. <Not valid apost>: If TAB, ->#3.2.
      3.7. <Not TAB>: If CTERM, return.
      3.8. <Not CTERM>: If CS <> CT, ->#4.
      3.9. <CS=CT>: Inc NT.  If NT<NTARGT, ->#3.1; else process for success.
             3.9.1. <Entry>: Increment NT.  If NT<NTARGT, ->#3.1.
             3.9.2. <NT=NTARGT: Success>: Set POSTN=NPOS; update ISOURC; return.

4. <CS <> CT>: Decrement NS to prior column; ->#2 to restart scan.
```

Table 12: Functional Block Outline #4 — Block #2 only

```
2. <FNAMFQ opened to LUNMOD>: Read thru, processing each "CALL " found.

      2.1. <Entry>: Get index of FNAMCR in MODNAM array, MNDXCR.

      2.2. <MNDXCR defined>: Read next line; if EOF on LUNMOD, ->#3.

      2.3. <Next line in CARD>: If comment card, ->#2.2; else set ISCOL=7.
             2.3.1. <Entry>: If 'C' or '*' in Col. 1, ->#2.2.
             2.3.2. <Not (C,*) in Col. 1>: If CTERM in Cols 2-6, ->#2.2.
             2.3.3. <Non-comment card>: Set ISCOL=7 for start of scan; drop.

      2.4. <ISCOL defined>: Call POSTN. If "CALL" not found, ->#2.2.
             2.4.1. <ISCOL defined>: Set ISTAT=POSTN(---,80,CARD,ISCOL,---)
             2.4.2. <ISTAT returned>: If ISTAT=0, ->#2.2.

      2.5. <"CALL" found>: Scan for  MDNAME & "(" following; if none, ->#2.2.
             2.5.1. <Entry>: Blank MDNAME, zero NC.
             2.5.2. <At Col. ISCOL>: If ISCOL>72, ->#2.2.
             2.5.3. <ISCOL<=72>: Get next C; inc ISCOL.  If ' ' or TAB, ->2.5.2.
             2.5.4. <Not ' ' or TAB>: If C="'", scan to next "'" & ->#2.4.
                      2.5.4.1. <Entry>: If C <> "'", ->#2.5.5.
                      2.5.4.2. <ISCOL defined>: If ISCOL>72, ->#2.2.
                      2.5.4.3. <ISCOL<=72>: Get next char in C; inc ISCOL.
                      2.5.4.4. <C defined>: If not "'", ->#2.5.4.2.
                      2.5.4.5. <C is "'">: ->#2.4 to resume scan.
             2.5.5. <Not "'">: If CTERM or "=", ->#2.2.
             2.5.6. <Not CTERM or "=">: If "(", ->#2.5.8.
             2.5.7. <Not "(">: Inc NC; set MDNAME(NC:NC)=C; ->#2.5.2.
             2.5.8. <C="(">: If NC=0, ->#2.4 to resume scan; else drop.

      2.6. <Name in (NC,MDNAME)>: If the same as FNAMCR, notify & ->#2.2.

      2.7. <MDNAME non-recursive>: Add to module list called by FNAMCR; ->#2.2.
             2.7.1. <Entry>: If MDNAME entered, skip ->#2.7.3.
             2.7.2. <Not yet entered>: Initialize it; drop.
                      2.7.2.1. <Entry>: Increment NMMODS.
                      2.7.2.2. <NMMODS inc>: Set MODNAM(NMMODS)=MDNAME; drop.
             2.7.3. <MDNAME in MODNAM>: Inc NMMCLD(MNDXCR); save in NMCMOD.
             2.7.4. <NMCMOD defined>: Set MODCLD(MNDXCR,NMCMOD)=MDNAME; ->#2.2.
```

Table 13: Final Functional Block Outline for Phase 1 of DOCALL — Part 1 of 4

```
MAXMOD         I     MAX No. MODules permitted
MAXCLD         I     MAX No. of MODules that may be CaLleD by a given module
NMMCLD  MAXMOD I     No. Modules CaLleD by this module
MODNAM  MAXMOD CH*15 Array of MODule NAMes encountered
MODCLD  MAXMOD,
        MAXCLD CH*15 Array of MODules CaLleD for ith module: MODNAM(i)
MEXIST  MAXMOD L     Array of flags indicating Modules EXIST
NMCOL          I     NuMber of COLumn positions available for storing names
IACOL   10     I     Integer Array of starting COLumn positions
MNDXCR         I     INDeX into MODNAM array for FNAMCR
MINDEX         I     INDEX into MODNAM for called MDNAME
FNAMCR         CH*15 NAMe of CuRrent File (module) being processed
NMCMOD         I     NuMber of MODules currently called by FNAMCR
MDNAME         CH*15 NAME of called MoDule
NMMODS         I     NuMber of MODuleS currently encountered
NMMPRC         I     NuMber of Modules currently PRoCessed
MAXLNG         I     MAX LeNGth of the MODule names.
LENMOD         I     LENgth of longest module name
MAXLIN         I     MAX length of LINE used for printing
LINE           CH*132LINE used to hold each line of printed output
RGDBLK         L     Flag to ReGarD BLanKs in comparison
RGDAPS         L     Flag to ReGarD APoStrophe-delimited by skipping
POSTN          Fun   Int POSiTion of start of target string, & iNc col.
CTERM          CH*1  Char TERMinating a FORTRAN statement on a line
C              CH*1  A general Character
CTARGT         CH*80 The TARGeT Char string used by POSTN
LUNMOD         I     LUN opened to MODule in FNAMFQ
LUNFIL         I     LUN opened to either output FILe or printer
IOUTDV         I     Int type of OUTput DeVice: 1(S), 2(P), 3(F)
-------------------------------------------------------------------------
Note: No imbedded '---' permitted between CALL & (.
-------------------------------------------------------------------------
PARAMETER (MAXMOD=500, MAXCLD=100)
DIMENSION MODCLD(MAXMOD,MAXCLD)
DIMENSION NMMCLD(MAXMOD), MODNAM(MAXMOD), MEXIST(MAXMOD)
DIMENSION IACOL(10)
INT    POSTN
LOG    MEXIST, RGDBLK, RGDAPS
CH*1   CTERM,  CAPSTR, C
CH*15  FNAMCR, MODNAM, MDNAME, MODCLD
CH*30  PTHNAM,
CH*60  FNAMFQ
CH*80  CARD,   CTARGT
CH*132 LINE
-------------------------------------------------------------------------
```

Table 14: Final Functional Block Outline for Phase 1 of DOCALL — Part 2 of 4

```
1. <Entry>: Initialization
      1.1. <Entry>: Define program constants; init counters.
             1.1.1. <Entry>: Define program constants.
                    CTERM  = '!'
                    CAPSTR = ''''
                    RGDAPS = .TRUE.
                    RGDBLK = .FALSE.
                    CTARGT = 'CALL'
                    NTARGT = 4
                    MAXLEN = LEN(MODNAM(1))
                    MAXLIN = 80
             1.1.2. <Constants defined>: Init counters.
                     NMMODS = 1
                     NMMPRC = 0
                     NMMCLD(1:MAXMOD) = 0
      1.2. <Conts & counters initd>: Process the LUNS.
             1.2.1. <Entry>: Define LUNs.
                     LUNMOD = 1
                     LUNIN  = 5
                     LUNOUT = 6
                     LUNFIL = 8
             1.2.2. <LUNs defined>: Open LUNIN & LUNOUT.
             1.2.3. <(LUNIN,LUNOUT) opened>: Open LUNFIL to user's choice.
                      1.2.3.1. <Entry>: Output to Screen or Printer or File?
                      1.2.3.2. <IOUTDV=1: Screen>: Set LUNFIL=LUNOUT; ->#1.3.
                      1.2.3.3. <IOUTDV=2: Printr>: Open LUNFIL->SYS$PRINT;->#1.3.
                      1.2.3.4. <IOUTDV=3: File>: Open LUNFIL->CALLLIST.DAT.
      1.3. <LUNs processed>: Get name of directory in PTHNAM.
      1.4. <PTHNAM defined>: Get name & length main module in (FNAMCR,LENMOD)
      1.5. <(FNAMCR,LENMOD) defined>: Set FNAMFQ = PTHNAM // FNAMCR.
      1.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             1.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#1.7.
             1.6.2. <FNAMFQ opened to LUNMOD>: Set MODNAM(1)=FNAMCR; ->#2.
      1.7. <FNAMFQ doesn't exist.  Notify user; ->#1.3.
```

Table 15: Final Functional Block Outline for Phase 1 of DOCALL — Part 3 of 4

```
2. <FNAMFQ opened to LUNMOD>: Read thru, processing each "CALL " found.
      2.1. <Entry>: Get index of FNAMCR in MODNAM array, MNDXCR.
      2.2. <MNDXCR defined>: Read next line; if EOF on LUNMOD, ->#3.
      2.3. <Next line in CARD>: If comment card, ->#2.2; else set ISCOL=7.
             2.3.1. <Entry>: If 'C' or '*' in Col. 1, ->#2.2.
             2.3.2. <Not (C,*) in Col. 1>: If CTERM in Cols 2-6, ->#2.2.
             2.3.3. <Non-comment card>: Set ISCOL=7 for start of scan; drop.
      2.4. <ISCOL defined>: Call POSTN. If "CALL" not found, ->#2.2.
             2.4.1. <ISCOL defined>: Set ISTAT=POSTN(---,80,CARD,ISCOL,---)
             2.4.2. <ISTAT returned>: If ISTAT=0, ->#2.2.
      2.5. <"CALL" found>: Scan for  MDNAME & "(" following; if none, ->#2.2.
             2.5.1. <Entry>: Blank MDNAME, zero NC.
             2.5.2. <At Col. ISCOL>: If ISCOL>72, ->#2.2.
             2.5.3. <ISCOL<=72>: Get next C; inc ISCOL.  If ' ' or TAB, ->2.5.2.
             2.5.4. <Not ' ' or TAB>: If C="'", scan to next "'" & ->#2.4.
                      2.5.4.1. <Entry>: If C <> "'", ->#2.5.5.
                      2.5.4.2. <ISCOL defined>: If ISCOL>72, ->#2.2.
                      2.5.4.3. <ISCOL<=72>: Get next char in C; inc ISCOL.
                      2.5.4.4. <C defined>: If not "'", ->#2.5.4.2.
                      2.5.4.5. <C is "'">: ->#2.4 to resume scan.
             2.5.5. <Not "'">: If CTERM or "=", ->#2.2.
             2.5.6. <Not CTERM or "=">: If "(", ->#2.5.8.
             2.5.7. <Not "(">: Inc NC; set MDNAME(NC:NC)=C; ->#2.5.2.
             2.5.8. <C="(">: If NC=0, ->#2.4 to resume scan; else drop.
      2.6. <Name in (NC,MDNAME)>: If the same as FNAMCR, notify & ->#2.2.
      2.7. <MDNAME non-recursive>: Add to module list called by FNAMCR; ->#2.2.
             2.7.1. <Entry>: If MDNAME entered, skip ->#2.7.3.
             2.7.2. <Not yet entered>: Initialize it; drop.
                      2.7.2.1. <Entry>: Increment NMMODS.
                      2.7.2.2. <NMMODS inc>: Set MODNAM(NMMODS)=MDNAME; drop.
             2.7.3. <MDNAME in MODNAM>: Inc NMMCLD(MNDXCR); save in NMCMOD.
             2.7.4. <NMCMOD defined>: Set MODCLD(MNDXCR,NMCMOD)=MDNAME; ->#2.2.
```

Table 16: Final Functional Block Outline for Phase 1 of DOCALL — Part 4 of 4

```
3. <EOF on LUNMOD>: Close it; get next module in FNAMCR; open & ->#2.
      3.1. <Entry>: Close LUNMOD.
      3.2. <LUNMOD closed>: Increment NMMPRC.
      3.3. <NMMPRC inc>: If NMMODS=NMMPRC, all through; ->#4.
      3.4. <NMMPRC < NMMODS>: Set FNAMCR=MODNAM(NMMPRC+1); update LENMOD.
      3.5. <(FNAMCR,LENMOD) defined>: Set FNAMFQ = PTHNAM // FNAMCR.
      3.6. <FNAMFQ defined>: If it exists, open to LUNMOD; init it; ->#2.
             3.6.1. <Entry>: Open FNAMFQ with STATUS='OLD'; if error, ->#3.7.
             3.6.2. <FNAMFQ opened to LUNMOD>: ->#2.
      3.7. <FNAMFQ doesn't exist>: Process this case; ->#3.2.
             3.7.1. <Entry>: Notify user.
             3.7.2. <User notified>: Set MEXIST(NMMPRC+1)=False; ->#3.2.

4. <All modules scanned>: Now output results, in hierarchial order.
      4.1. <Entry>: Compute NMCOL & IACOL for loading called module names.
             4.1.1. <Entry>: Set NMCOL = MAXLIN/(LENMOD+2) - 1
             4.1.2. <NMCOL defined>: Select each of NMCOL locs; EOLC=4.1.4.
             4.1.3. <I'th loc selected>: Set IACOL(I) = (LENMOD+2)*I
             4.1.4. <I'th loc processed>: Increment to next one.
      4.2. <(NMCOL,IACOL) defined>: Select each of the NMMODS modules; EOL=4.6.
      4.3. <I'th module name selected>: Branch on its MEXIST(I) flag.
      4.4. <I'th module doesn't exist>: Output onto a single line; ->EOL.
      4.5. <I'th module exists>: Process this case; drop.
             4.5.1. <Entry>: Set NMCMOD=NMMCLD(I).
             4.5.2. <NMCMOD defined>: Select each of NMCMOD names; EOLC=4.5.6.
             4.5.3. <J'th name selected>: If MOD(J,NMLOC)<>1, ->#4.5.5.
             4.5.4. <At start of new line>: Init LINE, depending on J.
                      4.5.4.1. <Entry>: If J=1, ->#4.5.4.3.
                      4.5.4.2. <J<>1>: Output existng LINE; blank LINE; ->#4.5.5
                      4.5.4.3. <J=1>: Blank LINE; drop.
                      4.5.4.4. <Blank>: Set LINE(1:LENMOD)=MODNAM(I); drop.
             4.5.5. <Start of LINE defined>: Set LINE(IACOL(J): )=MODCLD(I,J)
             4.5.6. <J'th name inserted>: Increment to next one.
      4.6. <I'th module output>: Increment to next module.

5. <All through>: Terminate.
```

References

Barry W. Boehm. *Software Engineering Ecnomics*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1975. Essays on Software Engineering.

Donald E. Knuth. Structured Programming with *go to* Statements. *Computing Surveys*, 6(4):291–291, December 1974.

Kenneth T. Orr. *Structured Systems Development*. YOURDON Press, 1133 Avenue of the Americas; New York, NY 10036, 1977.

Ronald C. Wackwitz and Jay Falck. *Structuring FORTRAN Modules by Functional Blocks: An Overview (and a COMP-AID Service)*. Technical Report, The COMP-AID Company, 513 Old Bear Creek Road; New Braunfels, TX 78132, February 1980.

Niklaus Wirth. Program development by stepwise refinement. *CACM*, 14(4):221–227, April 1971.

Niklaus Wirth. Program Development by Stepwise Refinement. In Edward Yourdon, editor, *Writings of the Revolution*, pages 99–111, YOURDON Press, 1133 Avenue of the Americas; New York, NY 10036, 1982. Selected Readings on Software Engineering.