Prepared for the
GEORGE C. MARSHALL
SPACE FLIGHT CENTER
Huntsville, Alabama

31 October 1973

Contract No.: NAS8-30376
MSFC No.: MSFC-DRL-389, Line Item No. 2
IBM No.: 73W-00327

SPACE LAB
SOFTWARE DEVELOPMENT AND INTEGRATION CONCEPTS STUDY REPORT

Volume II – APPENDICES

APPENDICES

APPENDIX	TITLE
A	MANAGEMENT CONCEPTS FOR TOP-DOWN STRUCTURED PROGRAMMING
B	COMPOSITE DESIGN: THE DESIGN OF MODULAR PROGRAMS
C	CHIEF PROGRAMMER TEAM MANAGEMENT OF PRODUCTION PROGRAMMING

APPENDIX A

MANAGEMENT CONCEPTS FOR TOP DOWN STRUCTURED PROGRAMMING

FSC 73-0001

MANAGEMENT CONCEPTS FOR TOP DOWN STRUCTURED PROGRAMMING

R. C. McHenry

TABLE OF CONTENTS

ILLUSTRATIONS

Table 3 Programmer productivity

Organization	Source lines per programmer day
Unit design, programming debugging, and testing	65
All professional	47
With librarian support	43
Entire team	35

professional work, which includes system design and documentation, but not librarian support. The third row includes all programming and librarian support. The last row presents the productivity of the entire team on the completed system (excluding requirements analysis).

Team experience and conclusions

The chief programmer team approach appears to be desirable for the type of project discussed in this paper because programmer efficiency was substantially improved. The quality of the programming was demonstrated by nearly error-free acceptance testing with real data, by successful operation after delivery, and by its acceptance by system users.

The information bank system was specified, developed, and tested during a 132 man-month project. The team, in this experiment, was a relatively experienced one, and it performed at an above-average level. Comparing results of this experiment with results for comparable projects that were organized more conventionally, we believe that chief programmer teams applying the methods described in this paper should probably be able to double normal productivity. In addition, the quality of the completed programs should be superior to conventionally produced programs in terms of lower levels of errors remaining, self-documentation, and ease of maintenance.

Another valuable experience of the chief programmer team approach was its manageability. The team had a lower than usual ratio of professional-to-support personnel. Because the number of people actually doing professional work was small, communications problems were significantly reduced. The chief programmer was more knowledgeable about the progress of the work than programming managers generally are because of his direct involvement in it and because the techniques used (particularly the Programming Production Library, top-down programming, and structured programming) made the status of the work highly

visible and understandable. This knowledge allowed both him and his management to react to problems sooner and more effectively than might have been the case had they been more detached from the work.

The relatively small size of the team made it highly responsive to change. The original functional specification went through six revisions, yet it was possible to adapt readily to major changes, even those occurring after programming was well along. Improved communication achieved through the consistent application of top-down programming, structured programming, and the PPL all contributed to team adaptability.

A functional organization was applied both within the team and to the project organization as a whole. Within the team, the functional distribution of work allowed team members to concentrate on those aspects of the job for which they were best equipped and most productive. At the project level, the functional organization allowed the chief programmer to concentrate on technical progress of the programming, both internally and in his relations with the system users. A very effective relationship was established between the chief programmer and the project manager, and no problems arose from the dual interface with the users—who fully understood the responsibilities of each of the managers. During a period when the chief programmer was off of the project, the backup programmer successfully ran the project.

The functional organization effectively broadened the range of career opportunities in the programming field by allowing senior programmers to continue to be productive in a technical capacity. Downward, the team approach offers programming related clerical opportunities to nonprogramming personnel. The team, as originally constituted, included a programmer technician for the clerical function, but two problems arose with this approach. The work did not require a programmer technician because the PPL procedures were well enough defined that no programming knowledge was required to operate it. Also, neither librarian support nor secretarial support became full-time jobs on the project. We, therefore, combined the two functions and trained a secretary to perform them. With two weeks of on-the-job training, the secretary was capable of acting as librarian by using the PPL. Combining the two jobs also worked well from a work load standpoint because when programming work was heavy then documentation was light, and vice versa.

The programming techniques and standards used by the team to enhance productivity and visibility also worked as planned. Top-down programming was similarly successful. System logic for one of the major programs ran correctly the first time and never

required a change as the program was expanded to its full size. This was helpful in debugging, since programs usually ran to completion, and the rare failures were readily traceable to newly added functions. Top-down programming also alleviated the interface problems normally associated with multiprogrammer projects, because interfaces were always defined and coded before any coding functions that made use of the interfaces.

The Programming Production Library run by the librarian-secretary achieved its objectives of removing many of the clerical aspects of programming from the programmer and of making the project more visible and, hence, more manageable. It also encouraged modularity of the programs and made top-down programming practical and effective.

Whereas the experiment was successful, there are still some unanswered questions and unsolved problems. Most obvious, perhaps, is whether the approach can be extended to larger projects. The best estimate at this time is that it probably can, but it needs to be tried. The general approach would be to begin a project with a single high-level team to do overall system design and nucleus development. After the nucleus is functioning, programmers on the original team could become chief programmers on teams developing major subsystems. The original team would assume control, review, validation, and testing duties and perform integration of the subsystems into the overall system. The process could be repeated at lower levels if necessary. It might appear that such a top-down evolution of the development process would increase the project time vis-à-vis the bottom-up approach. This is not necessarily true because of parallel development and integration, and it may take even less time. In any case, the risk should be substantially reduced because of the better visibility and management control in the team methodology.

A second major question concerns team composition and training. Because the team is a close-knit unit producing a large system at a faster-than-usual pace, close cooperation and good communication are essential. It is, therefore, desirable that team members be experienced professionals trained in the techniques described. Although a team may include one or possibly two less experienced programmers, larger teams would force the chief programmer to spend too high a percentage of his time in detailed training and supervision thereby reducing his own productivity. One solution may be to place newly trained programmers in program maintenance or in projects that are extending existing systems before placing them on teams that are developing new systems.

The selection of the chief programmer from among several candidates may be more difficult than was at first anticipated. The

chief programmer is responsible for team management and for technical representation of the project to a customer and to his own management. Therefore, management ability and experience are necessary qualifications. A chief programmer must also possess the creativity and drive to make significant technical contributions of his own and to assist other team members in making their contributions. This essential combination of skills rarely appears in the same individual. Thus the use of aptitude testing should probably be considered as part of the selection process. Potential chief programmers should of course first serve as backup programmers to obtain first-hand experience before taking on their own projects.

One final question that has frequently been asked is whether chief programmers are willing to accept the technical and managerial challenges of large projects with few people. Experienced chief programmers have responded to the challenges and have found that it leads to a degree of satisfaction that is hard to match.

To summarize, there is little in the chief programmer team organization and methodology that has not been previously tried. Laid bare, it is basically a functional organization of programming projects coupled with the use of tried and true tools to improve productivity and quality. It works well when it all fits snugly together and is applied in a consistent fashion over an entire project. Continuing evolution shows promise of making the programming production process more economical and more manageable.

CITED REFERENCES

1. *Chief Programmer Teams: Principles and Procedures*, Report No. FSC 71-5108, may be obtained from International Business Machines Corporation, Federal Systems Division, Gaithersburg, Maryland 20760.
2. C. Böhm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Communications of the ACM* 9, No. 3, 366–371 (May 1966).
3. K. Conrow and R. G. Smith, "NEATER2: a PL/I source statement reformatter," *Communications of the ACM* 13, No. 11 (November 1970).

1. Programming Strategy

Recent work has advanced the techniques involved in the program production process. This better way is based on the techniques of:

- o Structured Programming
- o Top Down Programming
- o Programming Support Libraries

The use of these techniques results in improvements in manageability, quality, productivity, and maintainability. A description of the techniques and how each contributes to these improvements follows.

1.1 Structured Programming

Structured programming is based on the mathematically proven Structure Theorem[1] which states that any proper program (a program with one entry and one exit) is equivalent to a program that contains as logic structures only:

- o sequence of two or more operations
- o conditional branch to one of two operations and return (IF y THEN b ELSE c)
- o repetition of an operation while a condition is true (DO WHILE)

Each of the three figures itself represents a proper program (see Figure 1). A large and complex program may then be developed by the appropriate nesting of these three basic figures within each other. The logic flow of such a program always proceeds from the beginning to the end without arbitrary branching. Where only these structures are used in the programming, there are no unconditional branches or statement labels to which to branch.

Figure 2 illustrates traditional code and structured code.

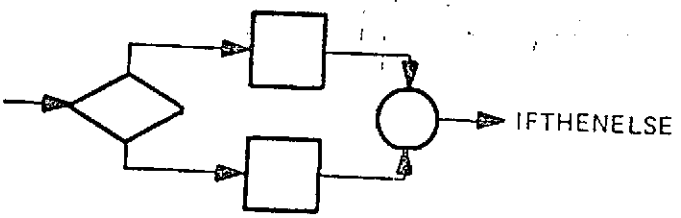
Structured programming reduces the arrangement of the program logic to a process like that found in engineering where logic circuits are constructed from a basic set of figures. As such, it represents a standard based on a solid theoretical foundation. It does not require ad hoc justification, case by case, in actual practice.

[1] Original form, Bohm and Jacopini, Comm ACM, May 1966, See also, Mills, Mathematical Foundations for Structured Programming, FSC 72-6012, February 1972.

Any proper program is equivalent to a program structure which contains, at most, the members:

Sequence of two operations:

Conditional branch to one of two operations and return:



Operation repeated while a condition is true:

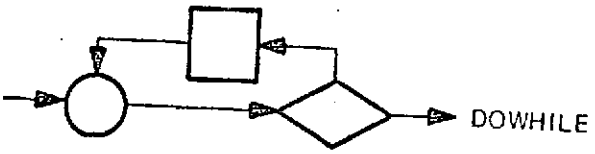


Figure 1. Structure Theorem

Several practices are included as a supporting part of the technique. For example, strict attention is paid to the indentation of the logic structures on the printed page so that logical relationships in the coding correspond to physical position on the listing. (See Figure 2). Thus, a pictorial representation of the logic is gained from the indentation. Another practice is that of segmenting code into reasonable amounts of logic that are each easily understandable. Each segment of code (whose internal operations may be any combination of the basic logic structures) must itself represent one of the basic logic structures. Thus, each code segment becomes a logical entity to be analyzed, coded and read at one time. (See Figure 3).

High level languages can be made almost totally self-documenting. For assembler level languages, macros provide the basic logic structures, giving these languages the readability and self-documenting attributes of higher level languages. The use of the basic logic structures coupled with indentation and segmentation rules, obviates the time consuming preparation of flow charts.

Simple extensions to the three basic logic structures are allowed. These do not affect the spirit of structured programming, but do result in more efficient use of computer time and storage.

1.2 Top Down Approach

Prior to actual implementation, functional requirements and software architecture will have been developed and described in the documented baselines of the definition and design phases.

Traditional software development has evolved as a bottom up procedure where the lowest level processing programs are coded first, unit tested, and made ready for integration (see Figure 4). Superfluous code in the form of driver programs is needed to perform the unit testing and lower levels of integration testing. Data definitions and interfaces tend to be simultaneously defined by more than one person and often are inconsistent. During integration, definition problems are recognized.

```
        IF p GOTO label q
        IF w GOTO label m
        L function
        GOTO label k
label m M function
        GOTO label k
label q IF q GOTO label t
        A function
        B function
        C function
label r IF NOT r GOTO label s
        D function
        GOTO label r
label s IF s GOTO label f
        E function
label v IF NOT v GOTO label k
        J function
label k K function
        END function
label f F function
        GOTO label v
label t IF t GOTO label a
        A function
        B function
        GOTO label w
label a A function
        B function
        G function
label u IF NOT u GOTO label w
        H function
label w IF NOT t GOTO label y
        I function
label y IF NOT v GOTO label k
        J function
        GOTO label k
```

TRADITIONAL

```
IF p THEN
   A function
   B function
   IF q THEN
      IF t THEN
         G function
         DOWHILE u
           H function
         ENDDO
         I function
      (ELSE)
      ENDIF
   ELSE
      C function
      DOWHILE r
        D function
      ENDDO
      IF s THEN
         F function
      ELSE
         E function
      ENDIF
   ENDIF
   IF v THEN
      J function
   (ELSE)
   ENDIF
ELSE
   IF w THEN
      M function
   ELSE
      L function
   ENDIF
ENDIF
K function
```

STRUCTURED

FIGURE 2. TRADITIONAL AND STRUCTURED CONTROL CODE

```
IF p THEN
   A function
   B function
   IF q THEN
      INCLUDE t-test
   ELSE
      C function
      DOWHILE r
        D function
      ENDDO
      CALL s-test
   ENDIF
   IF v THEN
      J function
   (ELSE)
   ENDIF
ELSE
   IF w THEN
      M function
   ELSE
      L function
   ENDIF
ENDIF
K function
```

t-test

```
IF t THEN
   G function
   DOWHILE u
     H function
   ENDDO
   I function
(ELSE)
ENDIF
```

s-test

```
IF s THEN
   F function
ELSE
   E function
ENDIF
```

FIGURE 3. Segmented Code

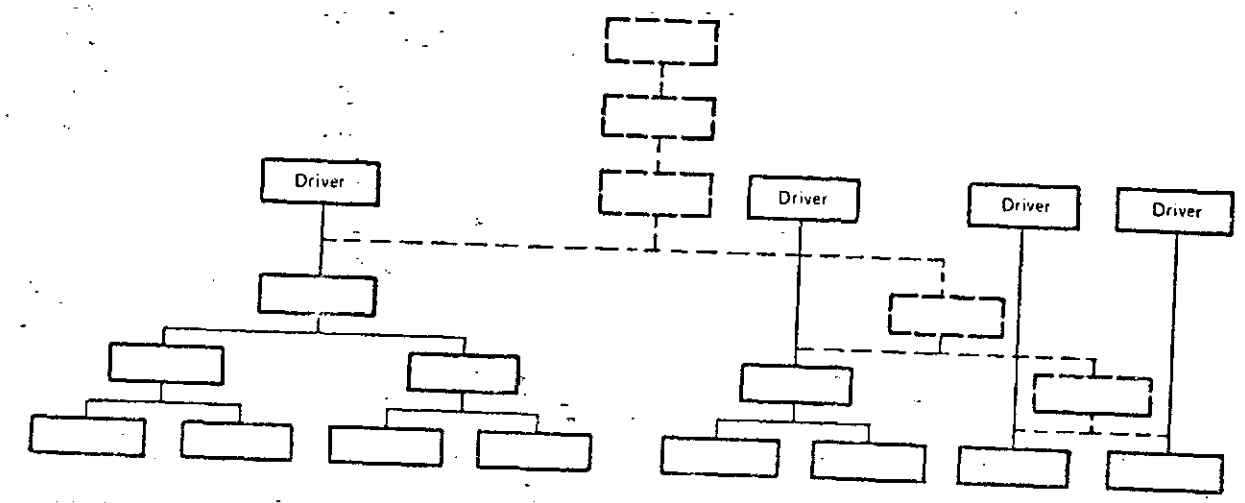
Integration is delayed while the data definitions and interfaces are correctly defined and the processing programs are reworked (and unit tested again) to accommodate the changes. It is often difficult to isolate a problem during the traditional integration cycle because of the large number of possible sources. Management control often is ineffective during much of the traditional development cycle because there is no coherent, visible product until integration.

The top down approach is patterned after the natural approach to system design and requires that programming proceed from developing the control architecture (interface) statements and initial data definitions downward to developing and integrating the functional units. Top down programming is an ordering of system development which allows for continual integration of the system parts as they are developed and provides for interfaces prior to the parts being developed (see Figure 4).

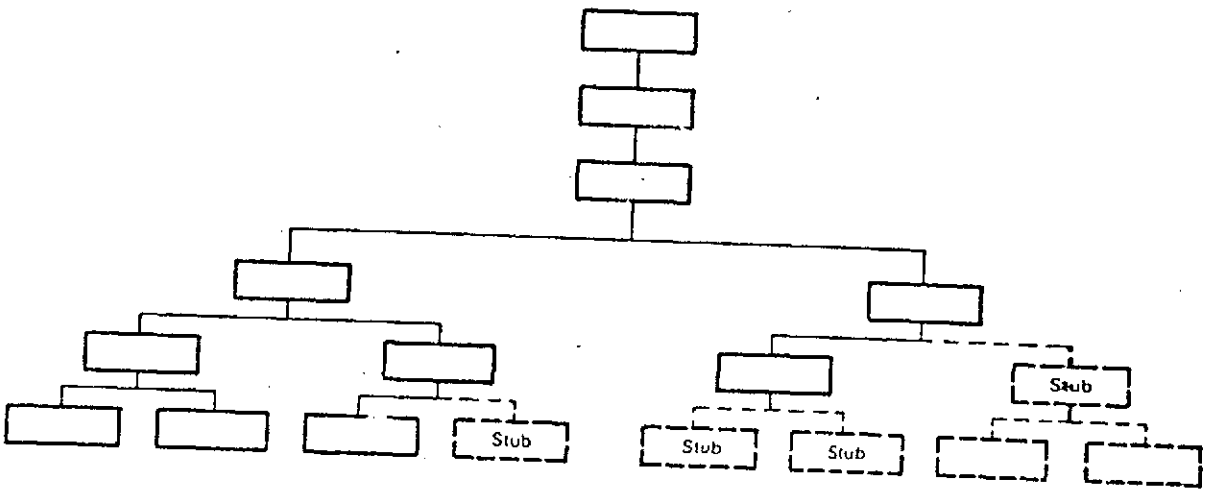
In top down, structured programming, the system is organized into a tree structure of segments. The top segment contains the highest level of control logic and decisions within the program, and either passes control to lower level segments, or identifies lower level segments for in-line inclusion. This process continues for as many levels as required until all functions within a system are defined in executable code.

Many system interfaces occur through the data base definition in addition to calling sequence parameters. The top down approach requires that the data base definition statements be coded and that actual data records be generated before exercising any segment which references them.

This approach provides the ability to evolve the product in a manner that maintains the characteristic of being always operable, extremely modular, and always available for successive levels of testing that accompany the corresponding levels of implementation. The quality of a system produced using the approach is increased, as reflected in fewer errors in the coding process. The act of structuring the logic calls for more forethought, and the uniformity and single entry, single exit attribute of the structured code itself contribute to the reduction in errors.



Traditional Bottom Up



Top Down

FIGURE 4. Approach Comparison

Due to the segmented nature of top down programming, the resulting system is extremely modular in function and logic structure. The quality increase realized by modularity extends itself into the documentation. The modular segments become natural units for documentation and for incremental learning of the system with a view toward maintenance and extension. Maintenance personnel begin learning about the system by reading the topmost segment and continuing down the various branches of the tree. The segments themselves are essentially self-documenting since the code in the segments is an elaboration of the structure logic of the original system specifications, and since the physical structure of the code on the printed page highlights that logic through indentation.

The approach introduces a significantly improved capability for management control of the software development effort by providing continuous product visibility. Since the developing system is undergoing continuous integration, the system status is accurately reflected through the contents of the system library; i.e., completeness is measured objectively in terms of how much of the system is operational. Managers can review the completed code to verify status and appraise the quality of the software product.

The approach alters or eliminates some of the traditional milestones usually associated with the program production process (see Figure 5). Probably the most obvious is the disappearance of any system integration period; it is no longer required since the system parts are continually being integrated as development proceeds.

Another area affected is that of initial documentation. In the past, given a set of functional system specifications, a detailed system design proceeding down through a set of detailed program specifications was written prior to coding. This has also been eliminated. Beginning with the design specifications, the various iterations on the detail of the design specifications are expressed in the code and not in prose. The developing system becomes the various levels of documentation, eliminating inconsistencies between programs and their documentation due to either lingual misinterpretation or temporal non-correspondence.

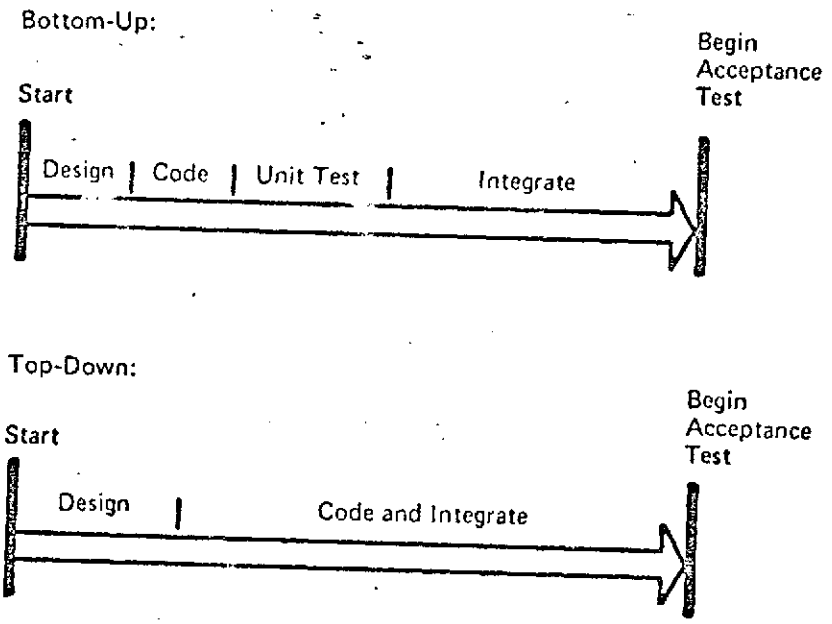


FIGURE 5. Milestone Comparison

Conceptually, top down implementation proceeds from a single starting point while conventional implementation proceeds from as many starting points as modules in the design. The single starting point does not imply that the implementation must proceed down the hierarchy in parallel. Some branches intentionally will be developed earlier than other branches. For example, user or other external interfaces might be developed to permit early training or hardware/software integration.

In systems with user interfaces, the user can interact with the (functionally incomplete) system much earlier in the development phase. This early interaction provides an opportunity to prepare user and operator guides top down (i.e., as user facilities are developed) and to validate the guides.

1.3 Programming Support Libraries

A programming support library is a set of office and computer procedures designed for use in a program development environment. The principal objective of the library is to provide constantly up-to-date representations of the programs and test data in both computer and human readable forms. The design of the procedures permits the clerical and recordkeeping operations associated with the programming to be isolated from the programmer. The library includes these facilities:

- Update Procedures to store and modify programming data.
- Version Control to permit one version of a system to be used as the baseline for the next. Since program segments unfold through the use of CALL's and INCLUDE's, it is critical that code under development not be accidentally called or included.
- Automatic Indentation of Source Listings to improve readability. This feature is a characteristic of library support, since listing should be possible independent of compilation.
- Office Procedures to provide visibility of the code on hand and to ensure successful operation of the library system.

- Creation of Dummy Program Segments to allow execution of a program throughout development as soon as the top-most segment is available.
- Library Recovery Procedures to provide backup and recovery of libraries or library segments.
- Housekeeping Procedures to allocate, catalog, restructure and maintain the libraries.
- Library Status Data to record unit ownership, size, specification and attributes which are needed to provide proper library maintenance.
- Automatic Listing of current library content.
- Program Directory and Cross Reference to show the hierarchical structure of units in programs and to identify macro usage, called programs and included code for maintenance purposes.
- Module and Function Status Data to provide visibility of progress to management. The contents and status of segments are made available. This data should be correlated to the functional capabilities of the program.
- Programming Activity Data recording, e.g., numbers of statements modified and delivered and numbers of transactions in the program segments.

The library provides a significant aid to test and evaluation in that the current operational software system code is centralized to avoid ambiguity of what is, and what is not, valid software as well as centralizing the valid test program code. At every point in time, the overall system library constitutes the current operational system. Consequently, considerable care is taken to see that new segments and data item definitions have been properly tested before they are added. This testing is carried out in development libraries, in which segments are created as needed, exist until the units have been testing and added to the system library, and are then purged. Considerably more leeway is permitted in adding to a development library than in adding to the system library. For example, if a segment references a data item for which it is not authorized, it cannot be added to the system library. Such an unauthorized access is permitted in a development library, although the user is warned that he has committed an apparent error.

2. Test Approach

The top down approach to testing and integration starts with the testing of the highest level system segment once it is coded. Since this segment will normally invoke or include lower level segments, code must exist for the next lower level segment. This code, called a program stub, may be empty, may output a message for debugging purposes each time it is executed or may provide a minimal subset of the functions required. These stubs are later expanded into full functional segments, which in turn require lower level segments. Integration is, therefore, a continuous activity throughout the development process. During testing, the system executes the segments that have been completed and uses the stubs where they have not. It is this characteristic of continuous integration that reduces the need for special test data drivers. The developing system itself can support testing because all the code that is to be executed before the newly added segments has previously been integrated and tested and can be used to feed test data to the new segments. For this reason, most problems are localized to the recently added code. As the new segments are tested within the developing system, the control architecture and system logic are also tested.

The simplest kinds of stubs are those represented by non-functional dummy code for debugging and testing. These simple stubs can be automatically created by support library facilities. Functional stubs, which may be compared to drivers, provide data to the higher level segment. These frequently used stubs may provide data through:

- fixed parameters
- simulation, e.g., using random numbers
- simplified or skeletal procedures

These stubs are generally simpler to prepare than traditional driver programs and often became part (e.g., the interfacing code) of the lower level segment.

Top down programming provides a basis for capturing performance data during the development cycle. By replacing each dummy with a timed sequence that utilizes the estimated length of time for that function, the developing system becomes a model. As dummy routines are replaced with working code, the performance results can be appraised against the performance objectives. In a similar manner, storage allocation can be modelled.

The testing cycle can be directly correlated with the phases of software development: definition, design, implementation and test (refer to Figure 6).

Test requirements identify the functions to be tested, specify the number of cases, the ranges and limits of data and describe the hardware and software environment. The test requirements specify the degree to which the product goals: function, interaction, performance, operability, and useability are evaluated. The system requirements provide definition for software development.

The test specification details the test design approach and test structure, and identifies the methodology and procedures for testing. The test specification is analogous in content to the software design specification. The functional part of the specification is subject to change control so that the specification tracks change to the software specification.

The design review tests the software specification compliance with system requirements and assesses implementational feasibility. The review also evaluates accuracy, compatibility with other software and hardware and compliance to standards. The specification is the baseline for implementation.

After the test specification has gone through a design review in conjunction with the software specification, test procedures are developed. A test procedure is the series of actions required to verify that a software function meets its specifications, doing what it is supposed to do and nothing else. These actions may include:

- Configuring (software and hardware)
- Conditioning the process

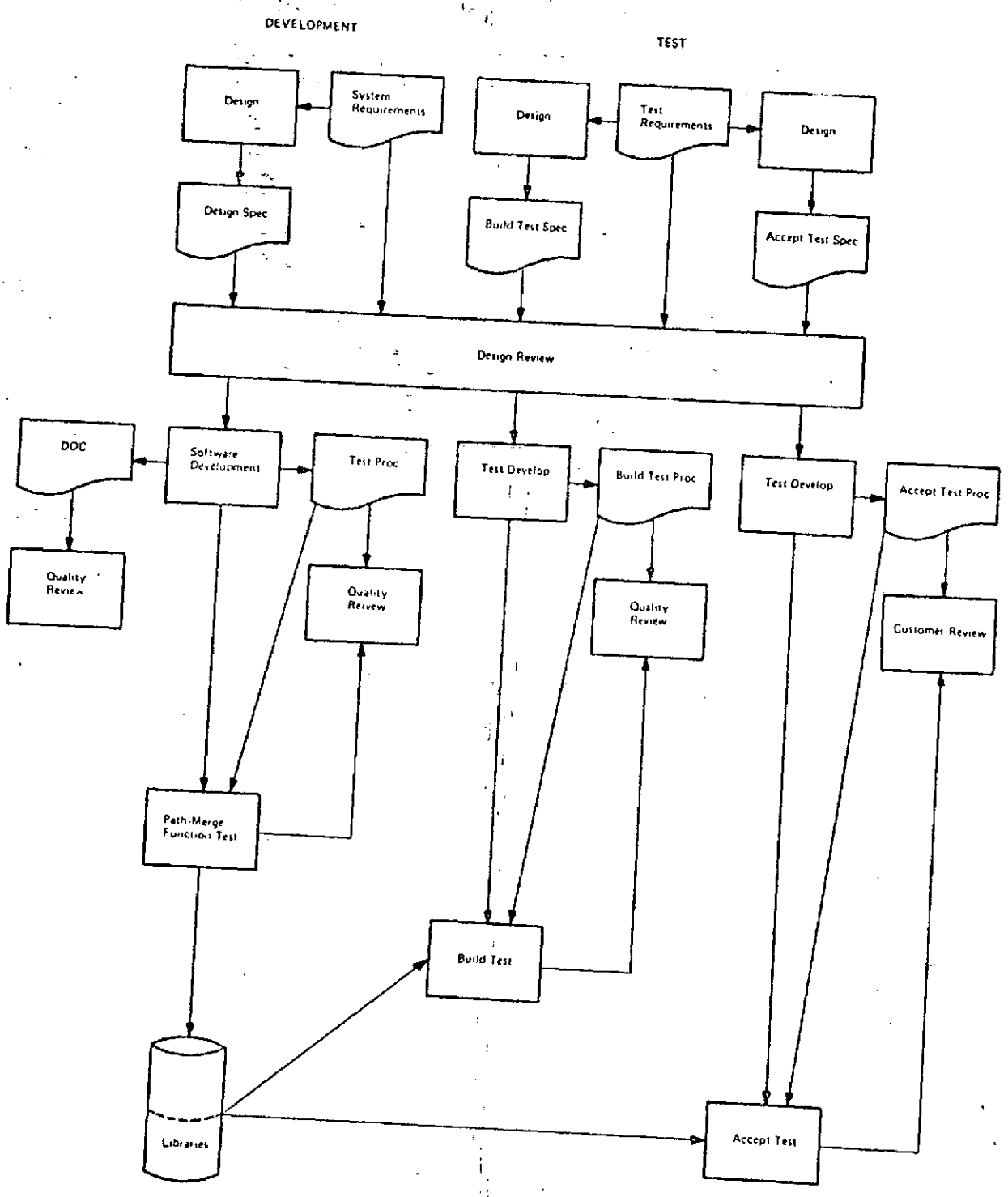


FIGURE 6. Test Phasing

- Introducing data
- Initiating execution
- Collecting intermediate/final data
- Displaying intermediate/final data
- Comparing actual to expected results
- Recording test results

The use of dummy code is viewed as special cases of configuring and conditioning software for testing.

The test tools, techniques, and procedure themselves must be validated.

Mandatory use of a management controlled system library promotes rapid resolution of interface problems and a steady increase in system performance and reliability. Control is obtained by requiring that an update to the system library be conditioned on proof of successful testing. This minimizes the likelihood of any software change getting into the system that would regress it and, therefore, preclude its release. The verification procedures are reviewed by the manager whose approval is required for update.

Path-merge testing combines the processing ability of two or more functionally related segments in order to test interfaces. The testing progresses as each lower level segment is added to previously checked out higher levels of the system.

Function testing verifies for each identified function that the baseline specifications have been met. This normally requires every line of code be executed. The desirable way to test a function is to interface it with the checked out portion of the system and test it by driving the total system. This type of testing accomplishes path-merge and function testing in one testing operation. When this is not practical, an individual driver is developed to test the function. Additional testing then must be done to assure that the function interfaces properly with the system.

Build testing determines the ability of the total software to perform in both nominal and non-nominal situations. This testing is the level after path-merge and function testing.

Performance testing determines if the product performs within an acceptable time frame and/or storage space as described in the specification. Simulation and modelling programs can be used to evaluate designs and predict the performance. Although these programs are on the periphery of what may be considered testing, they are valuable in design and product evaluation.

After the test execution, the test results are documented. Test results state how well the actual and expected results agree and explain any differences. The test specifications, test procedures and test results provide a complete record of the particular test for future reproduction. Conclusion and recommendation sections provide direction for proceeding to the next level of testing or for making changes to the test, as appropriate.

The documentation review is a test of the validity of the documentation. Each deliverable document is checked against all other related documents to insure consistency of terminology and technical accuracy.

Management Concepts

The methodology described in the preceding sections provides a basic level of technical control over software development and test. The project manager must maintain a balance of product (or service), resources, and schedule throughout the project. Tools exist for recording and reporting actual, estimated and budgeted resources and schedules. In most traditional software projects, the manager has limited visibility of the actual product, and thus has difficulty in maintaining the necessary balance. The difficulty is compounded by changes and problems. In top down software projects, the manager, through the libraries, has precise product visibility and control and high confidence that the product is truly what the tests to date indicate. In either approach, the manager has a risk that there are inadequacies in the design or the estimates for the remaining development and test activities.

The management concepts presented in this section enhance the visibility provided by normal configuration management procedures. These concepts are called programming discipline, testing discipline, and planning discipline. Each of these is keyed to the system library, which reflects the current status of the product being implemented.

The programming discipline assures that the system design reflected in specifications is consistent at all times with the software product. Figure 7 illustrates the iterative process of programming and testing of the software, following baseline design. All completed code must pass its specified testing requirements. If test results are satisfactory, the code is added to the system library. The system library is always available for independent product testing.

The testing discipline (Figure 8) ensures that all software interfaces properly and performs its intended functions prior to release. The testing discipline also provides an important measure of technical performance. Mandatory use of the controlled system library for all levels of testing ensures early resolution of interface problems. The results of tests form a basis for determining the current level of capability that exists, and product readiness for formal release to acceptance testing. Successful tests are reflected in the implementation plan.

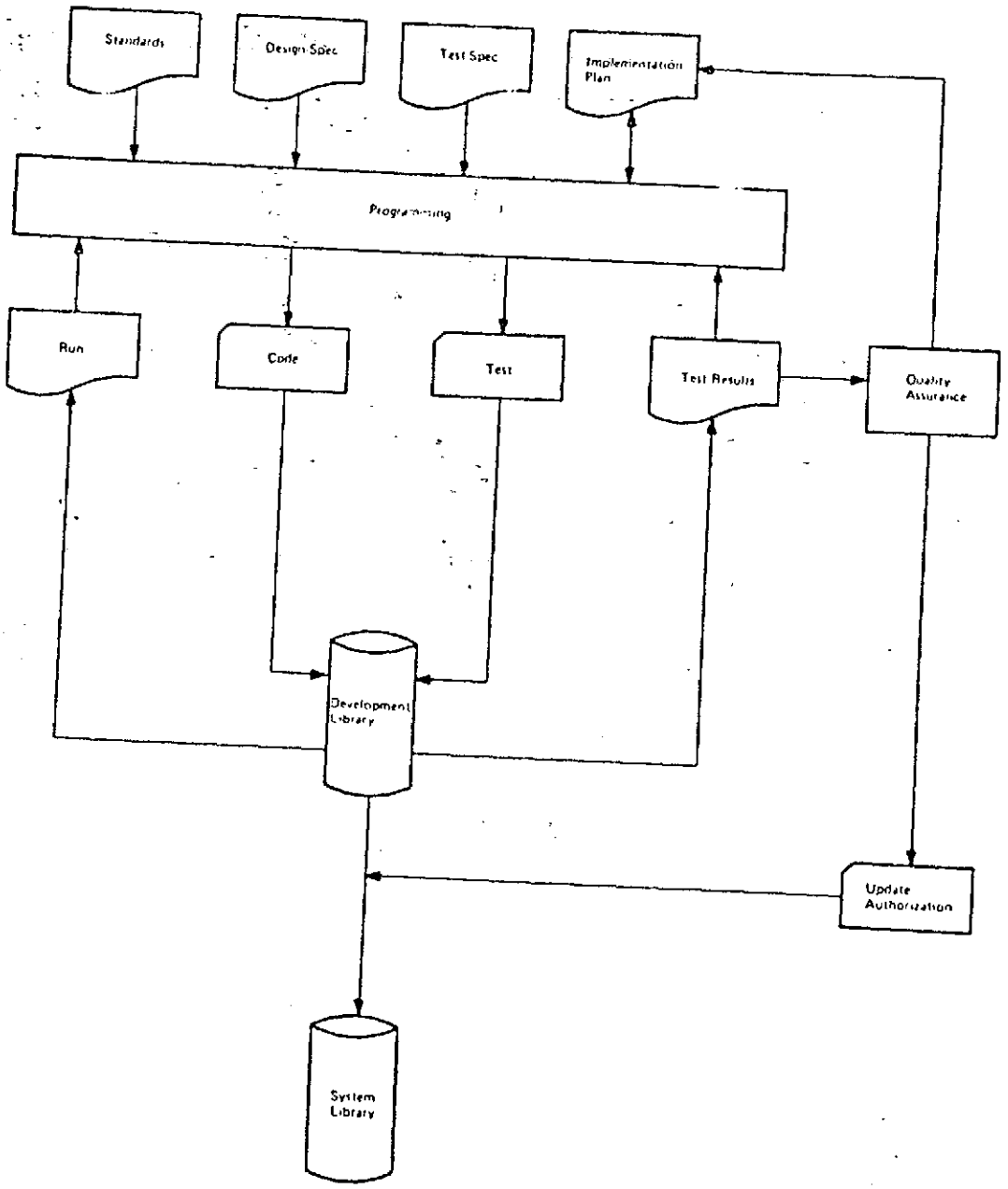


FIGURE 7. Programming Discipline

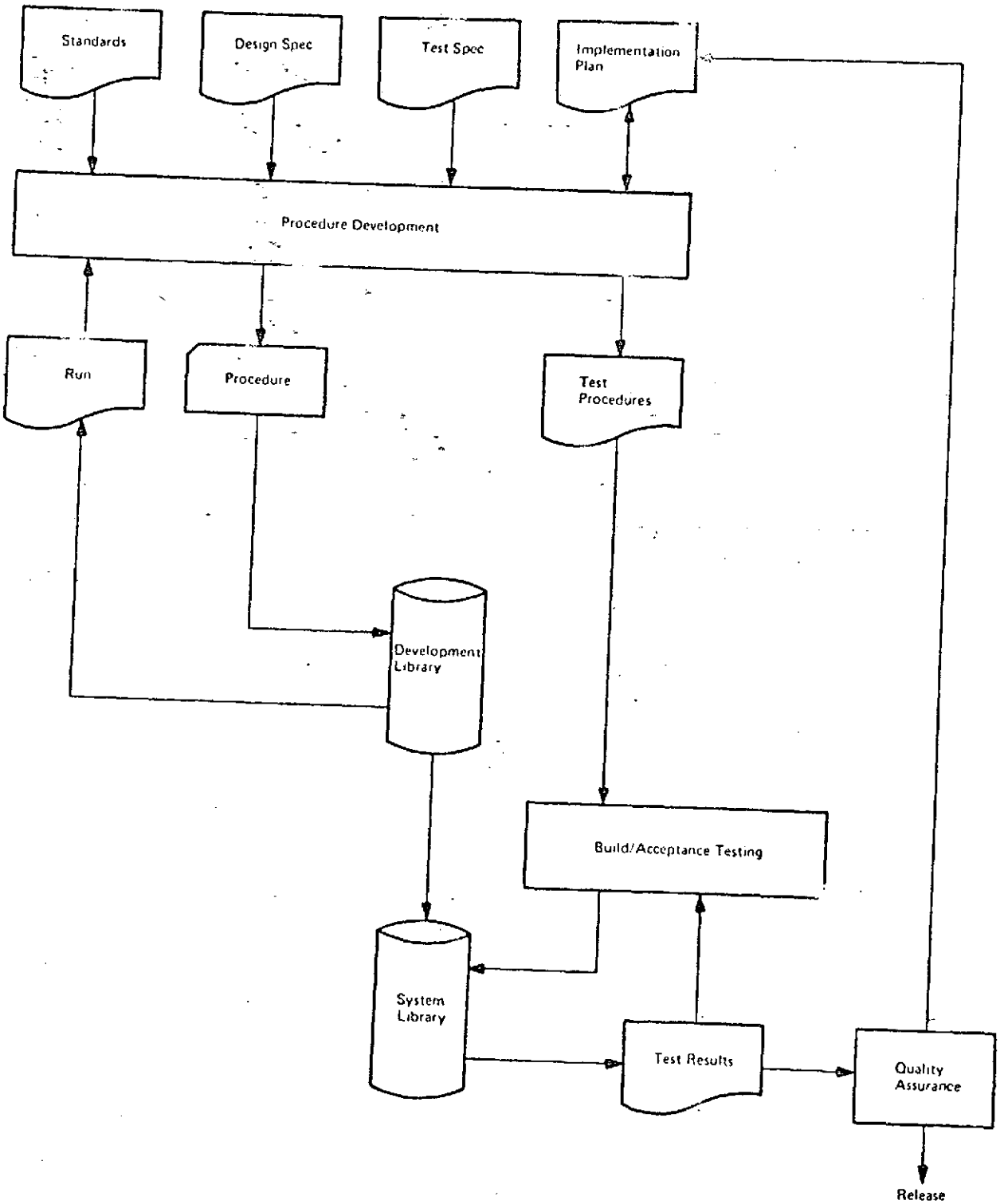


FIGURE 8. Testing Discipline

The planning discipline (see Figure 9) includes the following:

- o systematic, structured management review meeting to control plans, changes, problems and progress,
- o formal implementation plan for use as a record of progress made, plans, milestones, and actions taken by management,
- o controls for requirements and software changes,
- o provisions for documenting and correcting problems.

The interrelated procedures are the basis for management control at the technical working level.

A critical review of the technical status of software implementation and test is key to early assessment of situations impacting successful completion and delivery. In most project situations, a weekly cycle for review and reporting the technical status of implementation and test will assure adequate management visibility. Figure 9 illustrates the inputs to management review. Reviews would typically be conducted as described below.

Each senior manager conducts a meeting of his line managers and key technical personnel to review the implementation plan, discuss problems, status, schedules, and changes; and document recommendations for revising the implementation plan. The meeting prepares each senior manager for the program manager's meeting where decisions are reviewed and unresolved problems are addressed.

The program manager's meeting is attended by the senior managers, and contract, financial and configuration management personnel. This meeting considers plans, schedules, problems and change proposals affecting technical, financial or contractual performance.

The implementation plan is the basic management tool used in planning, reviewing and reporting the technical aspects of system development and testing activities. The implementation plan forms the basis of much of the discussion during the review meetings. At all times, the plan reflects the best technical judgment of the management team implementing and testing the software.

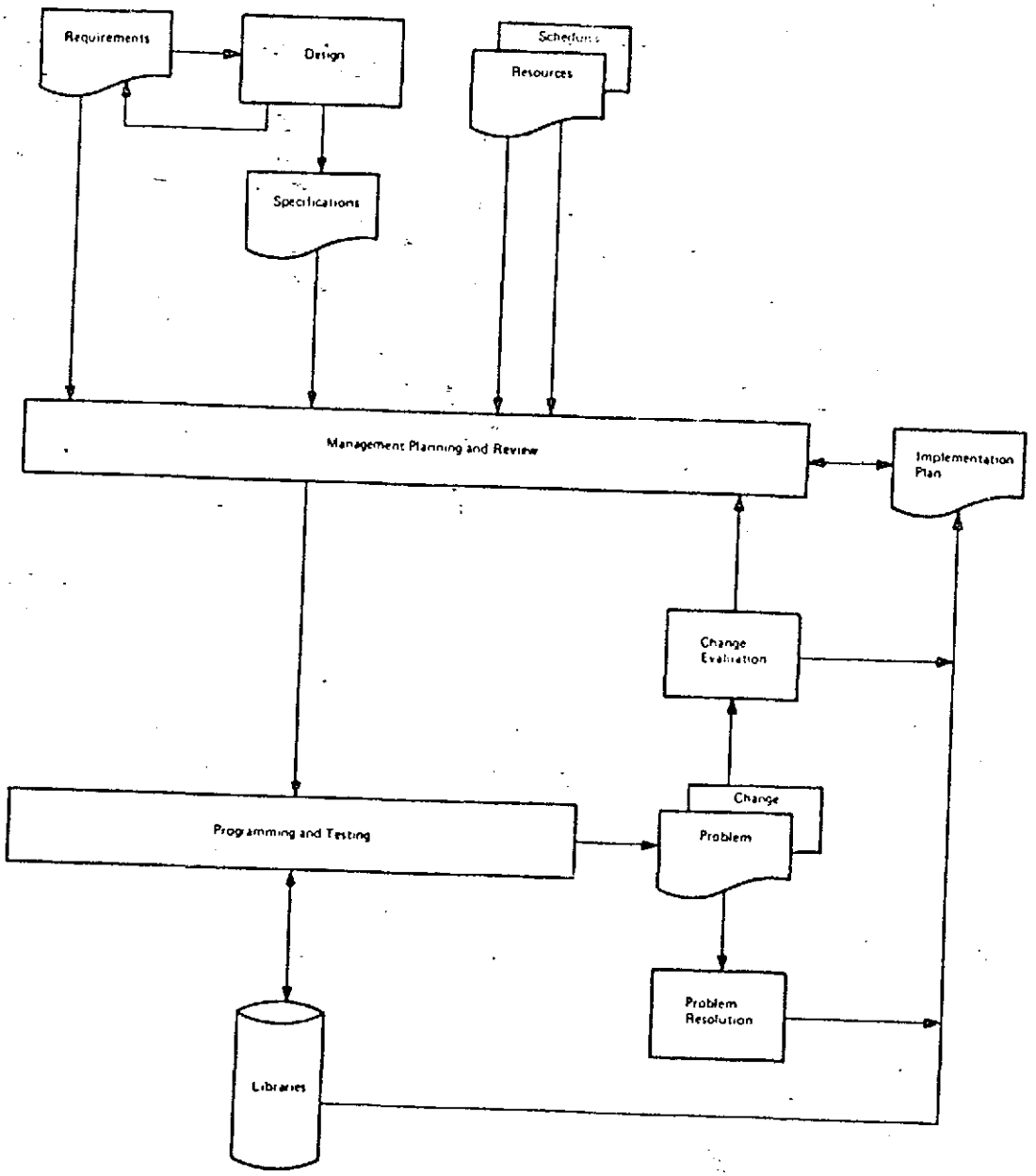


FIGURE 9. Planning Discipline

Programming and testing tasks which are reflected in the plan are created from three sources: baseline specifications, authorized changes and problems. Each of these sources must be carefully documented and controlled.

Specifications are required in all large software projects and form the baseline for implementation. The requirements (what) portion is usually subject to customer approval and change control. The design (how) portion normally is subject to internal project control.

Proper coordination and review of initial requirements and of revisions as they are made results in a more stable system. The objective is to firmly fix the responsibility for the acceptance of changes with management, which shields the individual programmer from severe pressures to accept and implement proposed changes. Adequate technical analysis and evaluation of all resources prior to making implementation decisions is ensured by coordinating all requirements changes with the management team. Unnecessary and costly changes and the unstable influence that their implementation would have upon the software may be avoided almost entirely by following the policy to commit implementation resources only after adequate review. Formal coordination and review of changes ensures adequate documentation and dissemination of the changes being accepted, as well as the proposed changes not yet acted upon, and the requests for modifications which have been rejected.

Once a segment has been added to the system library, anyone who experiences or observes a difficulty with that segment is charged with the responsibility to report the problem. Errors made and corrected prior to adding a segment to the system library are not recorded. Reported problems are placed in an active file of open problems and remain on the open list until appropriate closing action is approved through management review. Problems may be closed by: submitting modifications to correct the deficiency; stating an operation restriction on the use of the segment and the operational procedure to be followed; providing proper interpretation of requirements and intended use.

4. Documentation

The software documentation strategy is based on the self documenting nature of structured programs. The strategy exploits three documents: detailed requirements, software design and listings. Since the listings are the product of software development no new program documentation is produced after the design phase except the code itself.

A software development project has two major checkpoints: design review and acceptance test. There is documentation produced in the project phase which precedes each checkpoint. The analysis and design phase produces a design specification (typically including detailed requirements and software architecture) and, sometimes, a development plan, a standards manual; a test specification or their equivalent. The development phase produces program listings and program documentation (traditionally including narrative and flow charts) and, sometimes, user and operator guides and test procedures.

End item documentation (excluding listings and user and operator guides) which traditionally is prepared after the design phase consumes 5 to 15 percent of the programming project budget. The elimination of this documentation represent a significant reduction in project cost. The primary purpose of this documentation is to support program maintenance and modification.

The strategy simply is to eliminate high cost, low use program documentation. This step can be taken in traditional programming projects with moderate risk if impeding system testing and subsequent program maintenance. Because of the improved readability, structured programs are (nearly) self documenting. Thus, in projects employing structured programming techniques, program documentation can be eliminated with minimum risk. This strategy provides as final software documentation, the design specification and the listings. For the strategy to succeed, the design specification must be kept current through change control (see Figure 10) and must adequately document interface standards in addition to the architecture and functional structure.

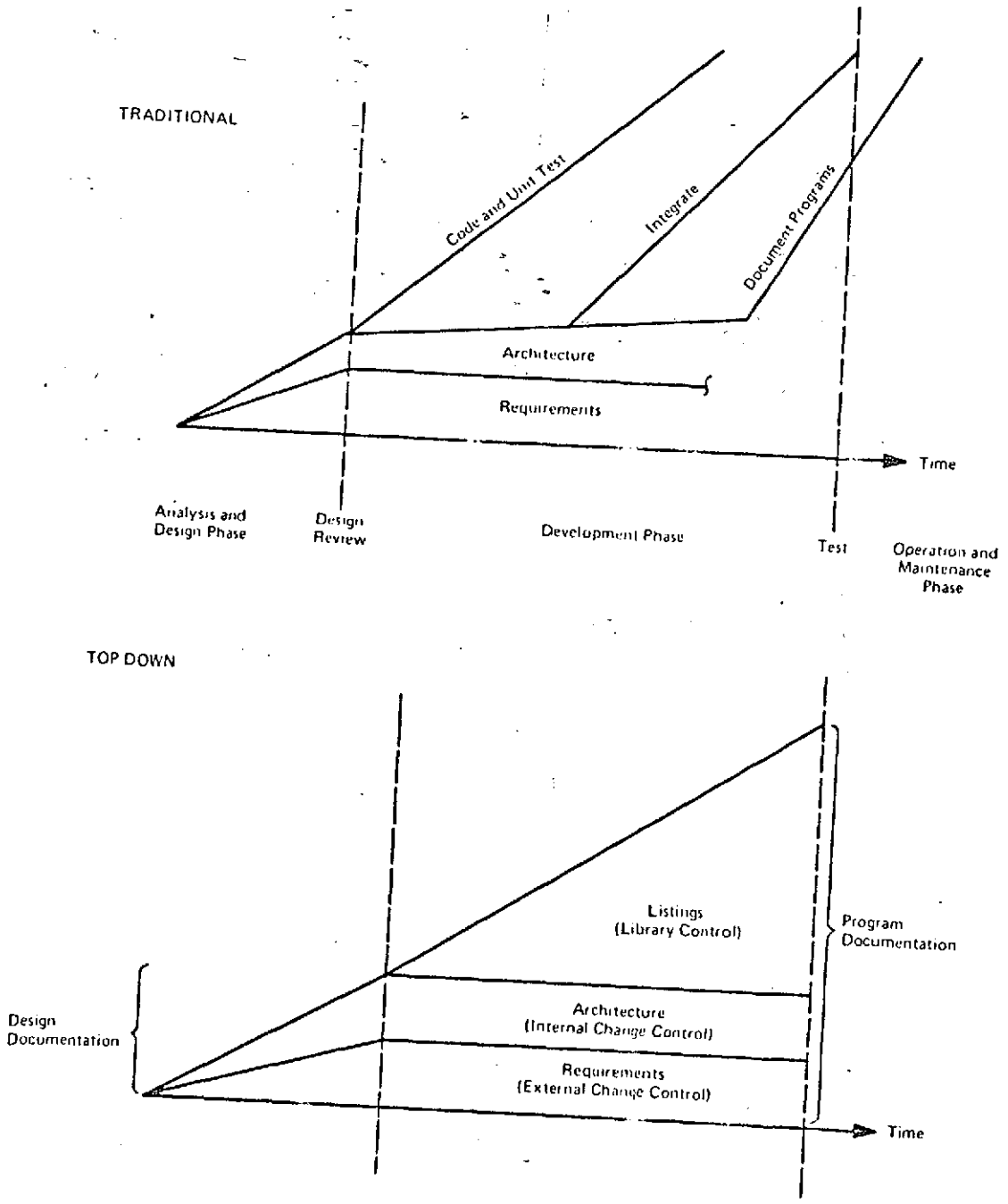


FIGURE 10. Documentation Comparison

Figure 11 illustrates the software development process and relates the documents to each phase. As the illustration suggests, the purposes of documentation are to discipline a subsequent activity and to record the results of an activity.

The system requirements provide a single, complete statement of the requirements that the software is to satisfy. Ideally, the system requirements are sufficiently complete and detailed to provide a basis for system design. However, in most projects a detailed requirements statement is part of the design documentation. As a consequence, some plan distinguish functional (requirements) and design specifications.

The test requirements identify the functions to be tested, and specify the number of cases, ranges and limits of data and hardware and software environment. The test requirements are specified at the same time as the system requirements. Test requirements discipline preparation of the test specifications.

The design specification contains two major parts: requirements and architecture. The parts are distinguished because the requirements are subject to customer (external) change control and the architecture is subject to project (internal) change control. This control provides a singificant benefit: an up-to-date specification which will be incorporated as part of the end item documentation.

The test specification specifies a design approach and test structure which will demonstrate that the software satisfies requirements. The test specification is similar in content and format to the software specifications. The test specification also is subject to change control.

The standards manual disciplines software development.

The implementation plan is a primary management control document.

The maintenance manual is the end item software product specification which provides a qualified programmer the information required to modify or maintain the software. This manual augments the up-to-date design specification with annotated machine listings. These listings provide completed deliverable documentation at the time of system acceptance.

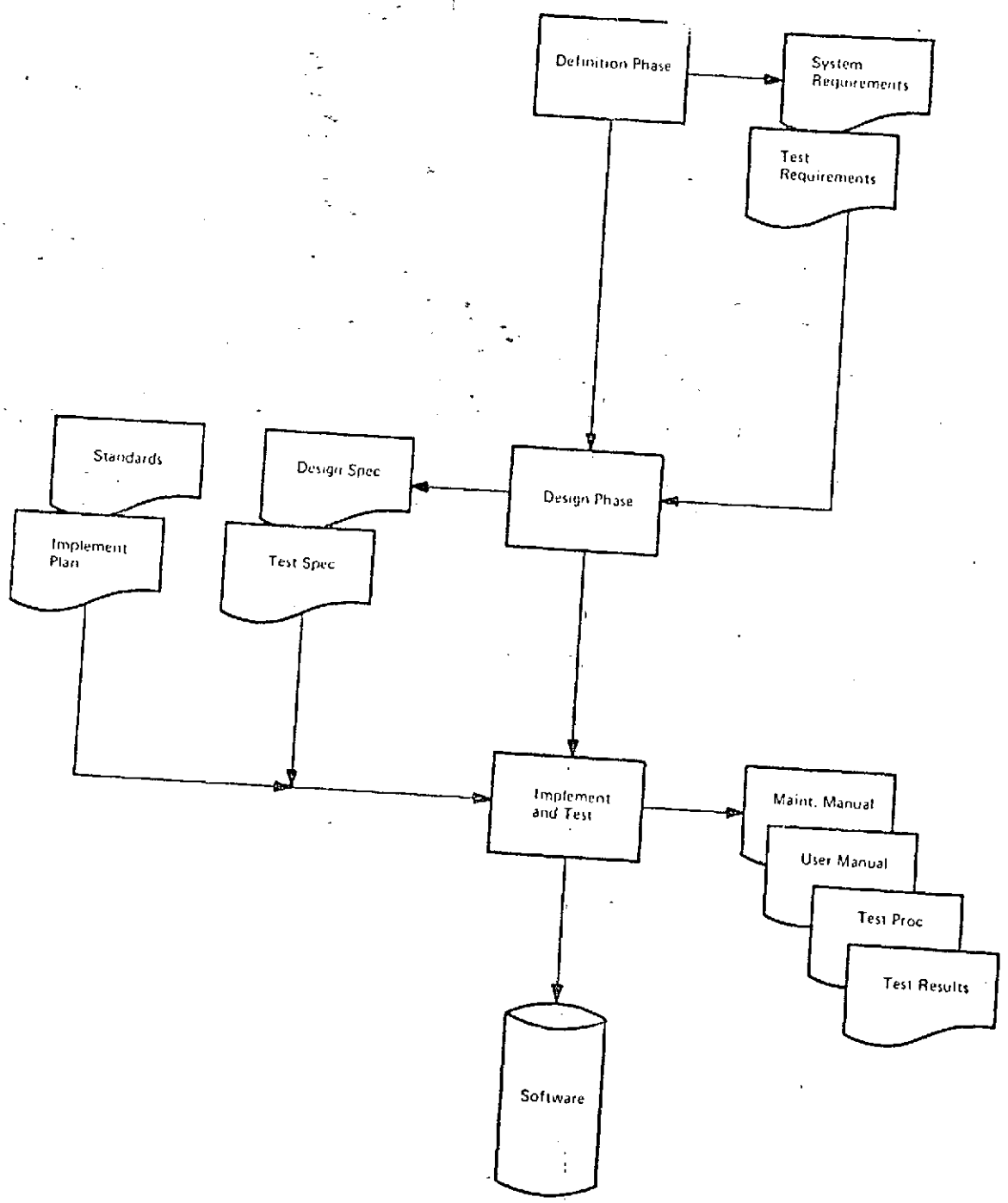


FIGURE 11. Document Relationship to Development Phase

The user and operator manuals provide two types of information. First, they provide the procedures to set up or initialize the system and the minimum equipment configuration including system generation constraints, parameters, default values, device assignments, etc. Second, the manuals cover the user techniques, messages and operator actions.

The test procedures are the detailed procedures, test data and expected results required to conduct a test. The controlled, up-to-date portion of the test specification is considered to be part of test procedure documentation.

The test results state how well the actual and expected results agree, and explain any differences. The test results and associated test specification and procedure provide a complete record of the particular test for future reproduction. Conclusion and recommendation sections provide direction for proceeding to the next level of testing, or for taking changes to the test, as appropriate.

APPENDIX B

COMPOSITE DESIGN:
THE DESIGN OF MODULAR PROGRAMS

TR 00.2406
January 29, 1973

Composite Design: The Design of Modular Programs

Glenford J. Myers

IBM

Technical Report

Poughkeepsie Laboratory

January 29, 1973

Composite Design: The Design of Modular Programs

by

Glenford J. Myers

ABSTRACT

Composite Design is a design technology used in the de
highly modular programs. It consists of a set of c
measures, analysis techniques, guidelines, notatio
terminology. By reducing complexity, it has a positive e
a program's quality, in terms of reliability, extensibil
cost.

IBM

Poughkeepsie Laboratory

CONTENTS

Part I BACKGROUND

Most computer programs are never designed; they are created on the coding pad. The blame for this falls on three parties:

1. Programming managers. Most managers are overly "output oriented". Since code is the largest part of the final product, they focus their attention on coding, diverting attention away from the more important task -- design.

2. Educators. Most programming schools, classes, texts, etc. teach coding. Program designing is almost completely ignored.

3. Programmers. Most programmers are totally unaware of good design strategies and techniques.

The purpose of this paper is to begin solving this problem, by defining a set of design measures, strategies, and techniques collectively known as "composite design". Part I sets the stage for this by discussing the realities of today's programming environment, discussing the three major factors of a programming effort, and defining some initial concepts, definitions, and notation for Composite Design.

During the initial development of this paper, the term "modular design" was used. However, several readers of the paper remarked that they had preconceived notions of modular design which they confused with the concepts in this paper. Hence, a new term, Composite Design, was chosen to represent these concepts.

1. PROGRAMMING TODAY

Perhaps the biggest problem facing programming today is the extreme difficulty and cost encountered in creating and maintaining large programming systems. An over-used term, "modularity", is often given as the answer to this problem.

To a large extent, modularity, when interpreted correctly, is the answer. "In particular, appropriate structuring of the system, its documentation, the project, its management, and all communication would greatly enhance maintainability and growth properties and extend the lifetime of large, complex programming systems."[1] Note the word "appropriate" in this quotation; this is key to many later concepts.

In the industry, there is a lot of experience and knowledge, both published and upublished, in the structuring of documentation, project organization, and project phases. Little thought is ever given to the structuring of the system itself. Hence, it appears that we can structure the programming development process, yet we can't structure the program.

One obvious argument at this point is that we do know a lot about programming. To a certain extent, this is true. We're reasonably good at designing the external aspects of a program, e.g., languages, performance constraints, human factors, file design, etc. We're also fairly proficient in the actual programming of a well-defined function. For instance, when faced with the task of programming a subroutine to convert binary numbers into decimal numbers, most programmers would have little difficulty in flowcharting, coding, and testing this subroutine using one or more techniques for coding, testing, etc.

Also, there is a lot of literature on the internal algorithms of a program or system, e.g., I/O buffering, paging, scheduling, sorting, memory allocation, and file searching.

To summarize, we know* how to

1. Design the external aspects of a system.
2. Design the internal algorithms of a system.
3. Design and code individual subroutines or programs within the system.

---

*I'm using "know" in a relative sense here. Certainly, our knowledge in these areas today is still quite limited.

Note the missing link. How did we get from step 1 to step 2 or to step 3. This missing link, the subject of this paper, is:

4. Design the internal structure of a system.

The missing link is better illustrated by an example which describes a "typical" programming development effort.

Support a rudimentary information retrieval system is to be developed. It will operate as an applications program, being multiprogrammed with other applications under the control of an operating system. The information retrieval system communicates with a group of terminals and a data base of abstracts.

The first step, involving several systems analysts, is to specify the external characteristics of the system. They specify the language seen by the terminal user. They also specify the data base design and certain performance constraints, such as terminal response time and data base search time.

The analysts go on to a second step, the internal design of the system. They design an algorithm to service the terminals and an algorithm to search the key words in the abstracts. Next, they hand their specifications to a programming group for implementation.

The programming group takes over, armed with a document containing specifications for the language, the file design, performance constraints, and several algorithms. They recognize that the first step for them is to define the modules* in their system, since having a "modular" design is apparently a good trait. They regard this step very informally** and as a nuisance, since it appears to be an obstacle to flowcharting and coding the system. They perform this step usually using a combination of the following strategies:

---

*I use "module" in a loose sense here, equating it to "subroutine". It will be more formally defined in section 3.

**I say "informally" for two reasons. First, I have never seen a project schedule that recognized a "structural design" step between external design and module design. Secondly, its significance is always overlooked. I have seen programmers criticize other programmers' external designs, detailed module designs, and code; I have never seen a modular structure criticized.

1. Draw an overall flowchart of the system, making each block in the flowchart a module.

2. Create an "initialization module", a "termination module", and several "processing modules".

3. Assign each programmer an arbitrary "piece" of the system, allowing each programmer to work out his own structure.

4. Create a module to handle "all input operations", another to handle "all output operations".

5. Look for identical sequences of operations throughout the system, creating a module for each sequence.

Once this step is done, the programmers' sigh with relief and perform their "real work", internal design, coding, testing, and (alas) debugging of each module. Finally, after several schedule slippages, a few design changes that unexpectedly affected almost every module, and some last-minute piecing and patching together, they get the system on the air. Over the next year, a series of modifications are requested. Each modification results in unexpected large internal changes. Finally, because the installation cannot afford paying a staff of programmers whose job is simply maintenance and modification of the system, the installation reluctantly stops all modifications. Now, the static information retrieval system cannot cope with the ever-changing needs of its users, so the users move elsewhere. End of a sad, but typical, tale.

The system died because no one recognized the need for "appropriate structuring of the system". Also, it will become obvious later on that the five strategies listed are poor design strategies.

Facts of Life

The following list is a set of generalizations about today's programming environment. Although most of them are obvious, it is worthwhile to think about each one before proceeding.

1. Programs have a long life. This is illustrated by the popularity of emulators on today's third generation systems. Programs written ten to fifteen years ago are still in operation.

   Another illustration is the number of releases or versions of programming products. For instance, OS/360 has had 21 official releases. Its smaller brother, DOS/360, has had even more.

As a corollary, we can say that programs never achieve stability. They never achieve freedom from bugs nor freedom from additions or changes.

2. Programmers spend a majority of their time correcting errors. If you observe a cross section of programmers (even those developing new programs), you will find that most of their time is spent in testing, debugging, and correcting errors.

3. More often than not, technical designs are determined based on subjective personal preferences. I have heard programmers say, "To design a general and extendible system, data should reside in a set of flexible control blocks." This statement has no technical merit and, as I shall later show, is far from the truth. In fact, the opposite of this statement, "... dat should not reside in a set of flexible control blocks" is closer to the truth.

4. We don't follow the principle of standing on others' shoulders. "Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way."[2]

   Picture the electrical engineer designing a new T.V. set. He certainly doesn't design each vacuum tube, transistor, and capacitor all over again; he relies on existing components. In fact, he normally designs on a much higher level, using "off-the-shelf" power supplies, oscillators, etc.

   This analogy certainly doesn't apply to programming today. In general, programming technologies haven't advanced to this level. Furthermore, programmers aren't encouraged to operate in this mode. In the majority of new programming systems, every single instruction is coded from scratch.

Acknowledgement

An acknowledgement in the beginning of this paper is appropriate. Some of the notation and terminology in this paper, and several of the ideas in many of the chapters are from class notes and the yet-to-be published book, Fundamentals of Program System Design, by Mr. L. L. Constantine. Mr. Constantine is a consultant with Information and Sciences Institute, Cambridge, Massachusetts and a part-time instructor at IBM's Systems Research Institute.

2. QUALITY, COST, AND TIME

Quality, cost, and time are the three principal factors in any programming effort. A programming project is an optimization problem where we attempt to optimize one, two, or three of these factors subject to possible constraints on the other factors. For example, a typical programming project might have the following goal:

> Produce a program, maximizing its quality, but subject to constraints on cost (budget) and elapsed time (schedule).

The factor of quality can be subdivided into factors of reliability, maintainability, modifiability, generality, usability, and performance. Cost can be subdivided into two major costs, labor and machine time. Since Composite Design has a positive effect on many of these, I will identify the relationships between each of these factors and Composite Design.

Since you will probably be critical of many of the following claims, I would suggest rereading this section after you read the remainder of the paper.

Reliability is a measure of the number of errors or "bugs" encountered in a program. Although no quantitative data is available, Composite Design appears to have a positive effect on reliability, since modular programs are less complex and testing of modular programs appears to be easier and much more straightforward.

Maintainability is a measure of the effort and time required to fix bugs in the program. Composite Design has no significant effect on maintainability. However, since I feel that errors can be isolated faster in a modular program, future measurements may turn up a positive relationship.

Modifiability is a measure of the cost of changing or extending the program in the future. Composite Design has a very positive effect on modifiability. This is substantiated in later sections.

Generality is a measure of the scope of functions that a program performs. Usability is a measure of the human factors of a program. Composite Design has no known effect on these factors.

Performance is a measure of the efficiency of a program, for example, in terms of execution speed and storage used. Composite Design can have a slightly negative effect on execution speed. This relationship is discussed in section twelve.

Composite Design appears to have a positive effect on the cost of developing a program. Although no quantitative proof is available yet, several trends are apparent:

1. Programmer productivity in implementing a program designed via Composite Design appears to be higher than normal. This is expected since productivity is inversely related to the complexity (interactions and dependencies) in a program. Composite Design creates program with "loosely coupled" (independent) parts, thus reducing the interactions and dependencies.

2. Design changes are cheaper because they normally affect only one part of the program.

3. The design of the program is very visible and usually understandable. This increases productivity and also eases the process of adding new programmers to the project.

4. Testing of the program can proceed in a straight-forward sequence of steps. In practice, this reduces the complexity of testing and allows testing progress to be measured.

These points are illustrated in sections nine and ten.

Composite Design, to date, has had no observable effect on the elapsed time of a project.

3. CONCEPTS AND DEFINITIONS

Today, "modularity" is a popular term. Often, terms such as "modular" are added to the names of programs, to the titles of books, etc. because, "to be modular is to be good." Unfortunately, "modularity" is a widely misused and ill-understood concept.

The first order of business is to define the term module. For now, we will not distinguish between "good" or "bad" modules, but simply define the basic characteristics of a module. A module is a group of one or more program statements with the following characteristics:

1. The statements are lexically together. That is, when viewing a listing of the statements, the statements are physically together on the listing.

2. The statements are bounded by identifiable boundaries (e.g., START and END statements).

3. The statements are collectively referenced by a name (the module name).

4. The statements can be referenced, by the module name, from any other part of the program.

Hence, we see that the module corresponds to structural entities is most languages, such as the subprogram and function in FORTRAN, the procedure in PL/I and ALGOL, and the CSECT in OS assembly language.

The purpose of a module (at least those modules containing executable statements) is to receive some input data, perform one or more transformations on the data, and return some output data. To depict this, we will use the following form:

```
CALL SQRT (A,B,C)
```

which means - execute module SQRT, where the data named A, B, and C are the input and output data.

For purposes of this paper, we will make the following assumptions concerning modules:

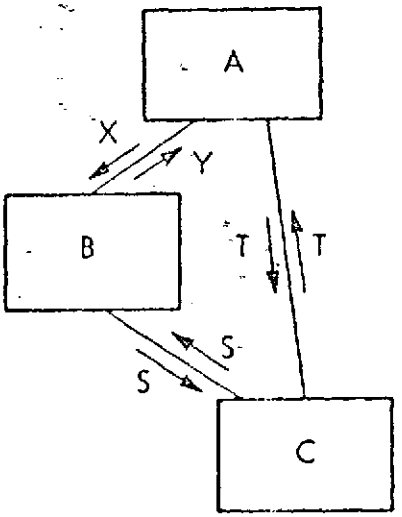
1. When a CALL statement is executed, execution in this module is suspended until execution of the called module ends.

2. When execution of a called module ends, execution of the calling module resumes with the statement immediately following the CALL statement.

3. When execution of a module ends, execution resumes in the calling module. More plainly stated, "all modules return to their callers."

Points one and two are simply popular conventions. Point three is a necessary condition in Composite Design.

Graphical notation plays an important part in Composite Design. The notation is illustrated in Appendix A. The following example will explain the basics of the notation.



From this diagram, we can determine the following:

1. There are three modules, A, B, and C.

2. Somewhere in module A, there are at least two CALL statements, one for module B and one for C.

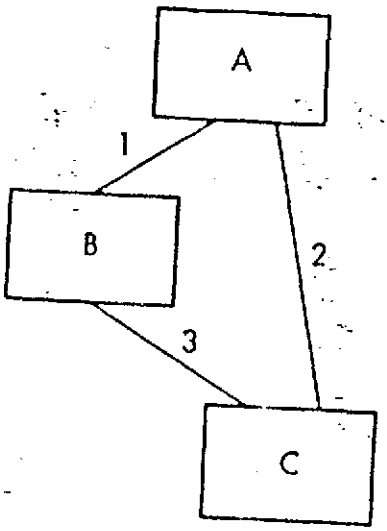
3. Somewhere in module B, there is at least one CALL statement for module C.

4. B receives an input of X and returns an output of Y. C receives an input of S or T, and passes S or T back as output, respectively.

5. B is subordinate to A. C is subordinate to both A and B.

Note that this type of diagram shows only structural relationships. It does not imply any procedural or algorithmic relationships. For instance, it does not tell us whether A calls B before it calls C, or vice versa, how many times A calls B, whether A calls B and C everytime A is executed, etc.

An alternate method for illustrating the parameters is shown below:



	IN	OUT
1	X	Y
2	T	T
3	S	S

A key, and often misunderstood, definition is the function of a module. A module's function is the transformation (input to output) that occurs when the module is called. In other words, a module's function is "what happens when that module is called".

Note that the function is related not only to the operations performed in that module, but also to the functions of any modules called by that module. When speaking of a module's function, the module should be viewed as a black box. That is, we shouldn't care how the module performs the function. In fact, we don't care whether the function is performed entirely within the module or whether the module calls other modules to perform the function.

Understanding this definition of the function of a module is crucial to understanding Composite Design.

A segment is a group of statements having some of the characteristics of a module. The statements are lexically together, bounded, and may or may not have a collective name (segment name). Modules are comprised of one or more segments, which are either placed in the module originally or are copied into the module at compile time. The concept of a segment is not used in Composite Design, although it is sometimes used in the later design and coding of individual modules (e.g., structured programming [3]).

The fan-out of a module is the number of unique modules that are called from that module. For example, in the previous diagram, the fan-out of module A is two. The fan-in of a module is the number of unique modules that call that module. In the previous diagram, the fan-in of module C is two.

Throughout this paper, I use two terms, the program and the problem. The program is what we're designing. The problem is the reason for the program. That is, the program is a solution to the problem (or class of problems).

Part II MEASURES OF MODULARITY

The most important consideration in program design is having a set of objective measures of the design. With such a set of measures, we can objectively evaluate the "goodness or badness" of a design.

The two most important measures in Composite Design are module strength and module coupling. Section four defines module strength, which is a measure of the "goodness" of an indivudal module. Section five defines module coupling, which is a measure of the interconnections and interrelationships among modules.

Section six defines several other less important measures of modularity.

4. MODULE STRENGTH

The optimal modular design is one in which the relationships among elements not in the same module are minimized. There are two ways of achieving this - minimizing the relationships among modules and maximizing relationships among elements in the same module. In practice, both ways are used.

The second method, maximizing relationships among elements in the same module, is the subject of this section. "Element" in this sense means any form of a "piece" of the module, such as a statement, a segment, or a "sub-function".

This measure, known as module strength or binding, is one of the the most important measures of a modular design. All other things being equal, a module with high strength is "good", and one with low strength is "bad".

The scale of strength, from highest to lowest, is shown below:

1. Functional
2. Communicational
3. Procedural
4. Classical
5. Logical
6. Coincidental

The scale is not linear. Functional binding is much stronger than all the rest and the bottom two are much weaker than all the rest

For each type of binding, we will define it, give an example, and try to rationalize why it is found at its particular position on the scale. We will see that high module strength has a positive effect on programming cost and on program quality (in terms of extensibility and maintainability).

COINCIDENTAL BINDING

Coincidental binding occurs when there is no meaningful relationship among the elements in a module. Coincidental binding is usually the result of one of the following situations.

1. An existing program is "modularized", by splitting it apart into modules.
2. Modules are created to consolidate "duplicate coding" in other modules.

As an example of the second situation, suppose the following sequence of instructions appears several times in a module or in several modules:

```
A = B + C

GET CARD

PUT OUTPUT

IE B=4, THEN E=0
```

A well-intentioned programmer may analyze the situation and decide to replace all such sequences with a CALL to module X, and then create a module X containing these four instructions.

Module X is now probably coincidentally bound, since these four instructions have no apparent relationships among one another. Suppose in the future we have a need in one of the modules originally containing these instructions to say GET TAPERECORD instead of GET CARD. We now have a problem. If we modify the instruction in module X, it is unusable to all of the other callers of X.

It is only fair to admit that, independent of a module's strength, there are instances when any module can be modified in such a fashion to make it unusable to all its callers. However, the probability of this happening is very high if the module is coincidentally bound.

LOGICAL BINDING

Logical binding, next on the scale, implies some logical relationship between the elements of a module. An example is a module which performs all input and output operations for the program or a module which edits all data.

The logically bound "edit all data" module would probably be implemented as follows. Assume the data to be edited are master file records, updates, deletions, and additions. Parameters passed to the module would be the data, and also a special parameter indicating the type of data. The first instruction in the module is probably a four-way branch, going to four sections of code, edit master record, edit update record, edit addition record, and edit deletion record.

Most likely, these four functions are intertwined in some way in the module, because the programmer took advantage of the fact that they exist in the same module. If the deletion record changes and requires a change to the edit deletion record function, we probably have a problem, since this function is intertwined with the other three.

In short, logical binding usually results in tricky code which is difficult to modify and in the passing of unnecessary parameters.

CLASSICAL BINDING

Classical binding is the same as logical binding, except the elements are also related in time. That is, the logically bound elements are executed sequentially in time.

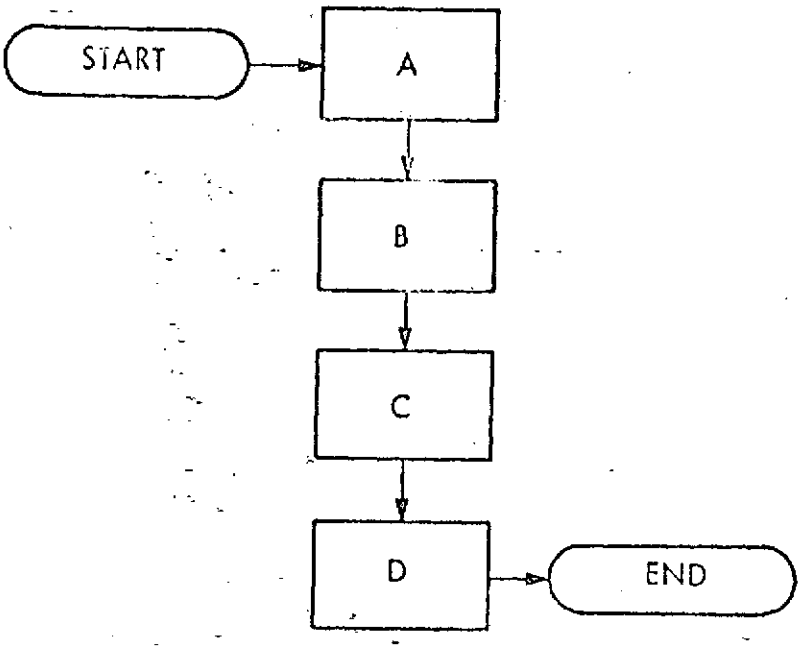
The best examples of modules in this class are the traditional "initialization", "termination", "house-keeping", and "clean-up" modules. Elements in an initialization module are logically bound because "initialization" represents a logical class of functions. In addition, these elements are related in time since the elements are executed together, sequentially in time (i.e., at "initialization" time).

Modules with classical binding tend to exhibit all of the disadvantages of strictly logically bound modules. However, classical modules are higher on the scale since they tend to be simpler, since all of the elements are usually executed at one time (i.e., no parameters and logic to determine which elements to execute).

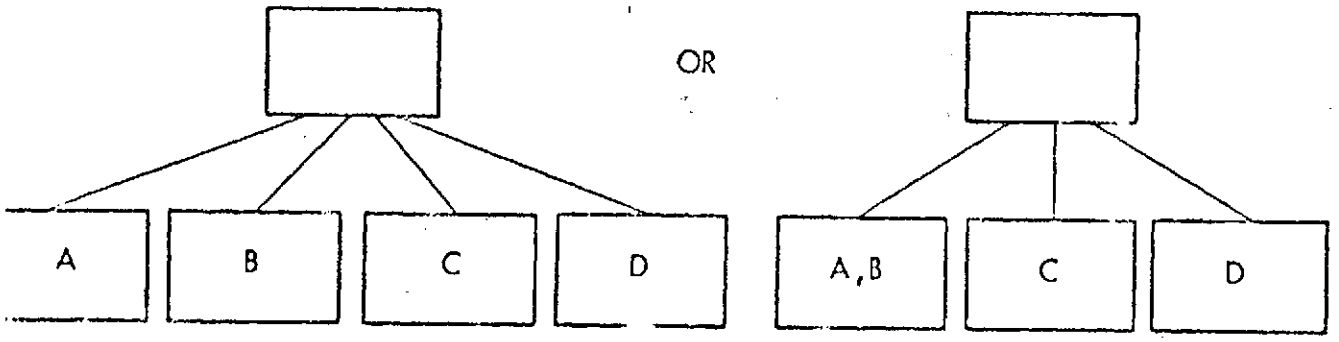
PROCEDURAL BINDING

Procedurally bound modules are modules whose elements are related in respect to the procedure of the program. Procedurally bound modules are the result of flowcharting the problem to be solved and then defining modules to represent one or more blocks in the flowchart.

The practice of designing by drawing an "overall flowchart" of the program usually results in modules with procedural binding, since flowcharting is procedure-oriented. To illustrate, assume the following flowchart represents the four sequential processes that make up a particular program:



If we were to use this to define the modules in the program, we would have procedurally bound modules (by definition). Typically, structures of procedurally bound modules for this program might look like this:



Procedural binding, although high on the strength scale because of a close relationship to the problem structure, is still far from the ideal - functional binding. The reason is that the procedural processes in a program are usually distinct from the functions in a program. Hence, a procedurally bound module can contain several functions or just part of a function.

COMMUNICATIONAL BINDING

A module with communicational binding is a module with procedural binding with an additional characteristic - the elements "communicate" with one another. That is, the elements in the module either reference the same set of data

or they pass data among themselves, e.g., the output of one element is the input of another element.

Consider the following modules:

A - update record in data base and record the record in audit trail

B - calculate new trajectory and send it to terminal

C - update record in data base and read next transaction

Module A has communicational binding, since the elements use a common set of data (the record). Module B also has communicational binding, since the output of the first element (the trajectory) is the input to the other element. Module C has procedural binding, since the elements do not communicate.

Communicational binding is higher on the scale than procedural binding since the elements in a module with communicational binding have a stronger "bond". That is, not only are they procedurally bound, but they reference the same data.

By now, you may have observed that a module can partly or wholely have the characteristics of more than one strength. If a module completely exhibits several types of strengths, we classify it by the higher strength. For instance, a module with communicational binding also has, by definition, procedural binding. However, we classify it by the higher strength, communicational binding.

A module which partially exhibits several strengths is classified according to the lower strength. For instance, if a module has three elements, all of which have procedural binding and two of which have communicational binding, the module has procedural binding. A module with part classical binding and part procedural binding (e.g., "read all input transactions and all master records and then print report headings") is classified with the lower strength, classical binding.

FUNCTIONAL BINDING

Functional binding is at the top of the strength scale. In a functionally bound module, all of the elements are related to the performance of a single function.

A question that always arises at this point is - what is a function? In mathematics, Y=F(X) is read "Y is a function F of X." The function F defines a transformation or mapping of the independent (or input) variable X into the dependent (or output) variable Y. Hence, a function describes a

transformation from some input data to some output data. In terms of programming, we broaden this definition to allow functions with no input data and functions with no output data.

In pratice, the above definition does not clearly describe a functionally bound module. One hint is that if the module does not fit the descriptions of the other types of binding (communicational, procedural, classical, logical, coincidental), it is probably functionally bound. Another way of learning to recognize functional binding is simply to use Composite Design. After finishing this paper, you should have a clear concept of functional binding.

Examples of functionally bound modules are:

Compute square root

Obtain random number

Write record to output file

Delete record from master file

The first module, Compute Square Root, is a function with an input and an output (square root of the input). The second module, Obtain Random Number, is a function with an output, but no input. The last two, Write Record to Output File and Delete Record from Master File, are functions with an input argument, but no output argument.

A useful technique in determining whether a module is functionally bound is writing a sentence describing the function (purpose) of the module, and then examining the sentence. The following tests can be made:

1. If the sentence is a compound sentence, contains a comma, or contains more than one verb, the module is probably performing more than one function, therefore, it probably has procedural or communicational binding.

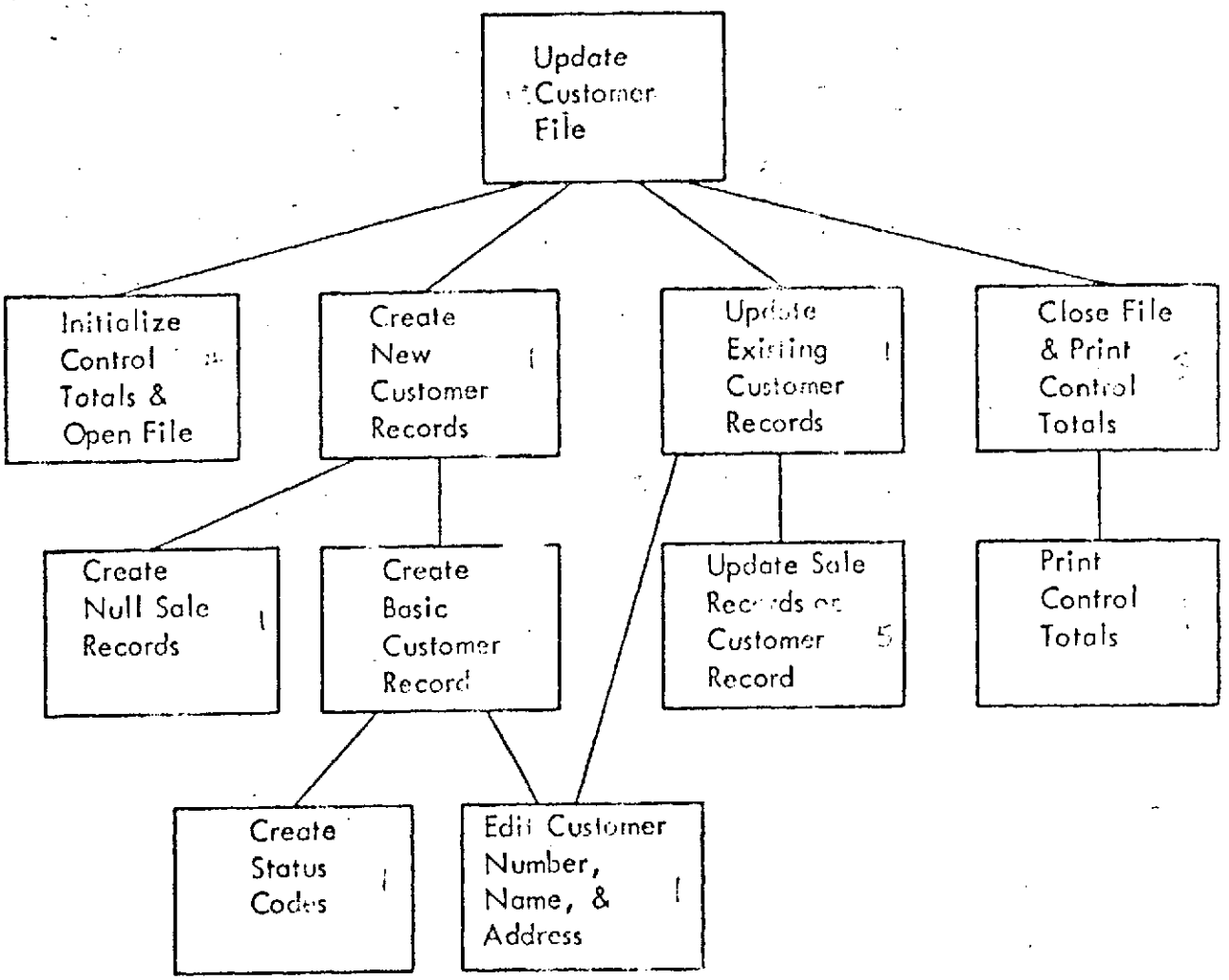
2. If the sentence contains words relating to time, such as "first", "next", "then", "after", "when", "start", etc., then the module probably has procedural binding.

3. If the predicate of the sentence doesn't contain a single specific object following the verb, the module is probably logically bound. For example, Edit All Data has logical binding; Edit Source Statement has functional binding.

4. Words such as "initialize", "clean-up", etc. imply classical binding.

The following diagram is a structural diagram of a typical (and actual) program. The purpose of the program (i.e., the function of the "top" module) is to update a customer file. Input to the first module is a customer record (e.g., number, name, address, status, and data on sales). The customer file contains customer records. Each customer record is followed by any number of sale records, containing data on sales to that particular customer. The program will create a new customer record or update an existing customer record.

The reader is invited to carefully inspect each module and determine their strength. Your results can then be compared with the analysis on the following page.



The analysis of this structure follows:

1. Module "Initialize Control Totals and Open File" represents classical binding.

2. Module "Update Sale Records or Custoner Record" has logical binding since it performs a class of logically related functions (updating records).

3. Module "Close File and Print Control Totals" has procedural binding since its elements, close file and print totals, are related only through the procedure of the program.

4. The other modules appear to have functional binding.

MODULE : COUPLING

There are two major measures of modularity. The first, module strength (binding), described in the previous section, is a measure of the binding among the internal elements of a module. The second measure, coupling, is a measure of the relationships among modules.

Coupling is a measure of the independence of modules. Since a highly modular design is achieved by maximizing the relationships among the elements of a module and minimizing the relationships among modules, the scale for coupling is inverse to the scale for strength. That is, we try to achieve high strength and low coupling.

The scale of coupling, from lowest coupling (best) to highest (worst), is:

1. data coupling
2. common coupling
3. control coupling
4. external coupling
5. content coupling

As the scale of strength, the coupling scale is not linear. Data coupling is very low, control coupling and external coupling are close to mid-range, and content coupling is very high. The placement of common coupling on the scale varies, depending on how it is used.

Following the pattern of the previous section, we will define each type of coupling, give an example, and explain why it sits where it is on the scale.

CONTENT COUPLING

Two modules are content coupled if one module makes a direct reference to the contents of the other module. This occurs in the following situations:

1. One module modifies a program statement in another module.
2. One module refers to non-externally declared data in another module. An example of "non-externally declared data" is a data element in a PL/I module that does not have the EXTERNAL attribute. Thus, a reference to data in another module where the symbolic name of the data was not resolved by a preprocessor, such as a linkage editor, implies content coupling.

3. Two modules share the same contents. This can occur when the statements of one module lie physically within another module or when two modules physically reside in one "compilable entity" (e.g., two CSECTS in the same "module" in OS Assembly Language).

It should be obvious that modules that are content coupled are very dependent upon one another and that a seemingly innocent change in one module can easily cause the other module to malfunction.

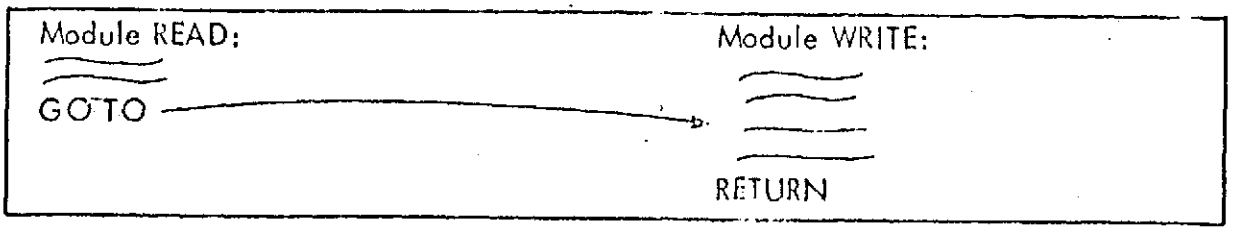
In situations one and two above, one module is dependent on actual displacements within the second module. Hence, almost any future change to the second module will require a change in the first module. Also, a significant change in one module, such as the use of a new algorithm or a change in data attributes or format, may require an extensive design change of the entire program.

Although situation three does not necessarily imply situations one or two, we can show that it sets up a very good "ambush" to allow the programmer to easily create situation one or two. Suppose that two modules, READ-FROM-TERMINAL and WRITE-TO-TERMINAL, are created and exist in one "compilable entity." Suppose, also, that they started out containing two unique sets of program statements and data.

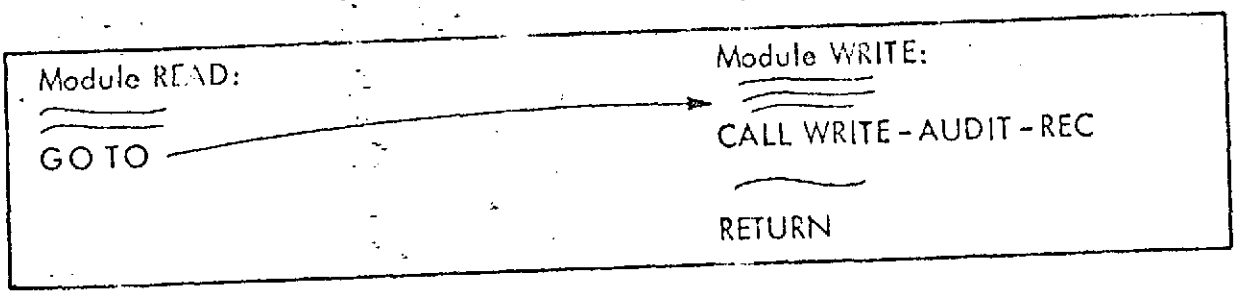
At a later point in time, while a programmer is modifying the input/output statements in the two modules, he notices that most of the input/output statements in the two modules are identical. In a move to "economize", he removes the statements from one module and simply branches into the other module. (He could do this because they were both in the same "compilable entity".)

ORIGINAL

NOW



So far, the two modules still operate correctly, but they are tightly coupled. Now, a new programmer is asked to change module WRITE so that it writes some data to an audit trail before it writes to the terminal. He writes a WRITE-AUDIT-REC 3D module and, upon examining module WRITE, inserts a CALL instruction as follows:



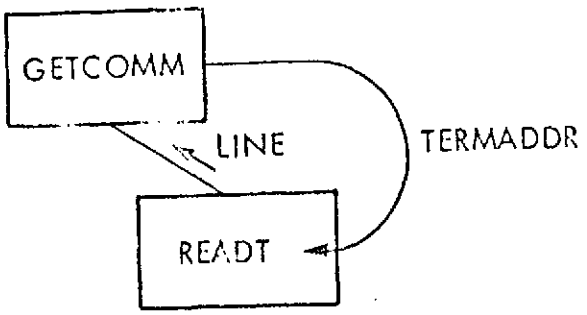
He has now created a bug in the program, since the execution of module READ now also causes an audit record to be written.

EXTERNAL COUPLING

Two modules are externally coupled if one module makes a reference to an externally-declared symbol in the other module. For instance, a PL/I module referencing a symbol with the EXTERNAL attribute, or an assembly language module containing a "V-con", are externally coupled with another module.

Since external coupling implies high coupling (remember, low coupling is what we're shooting for), and yet external coupling is a common programming practice, it's worthwhile to dig more deeply into this type of coupling.

Consider the following case:



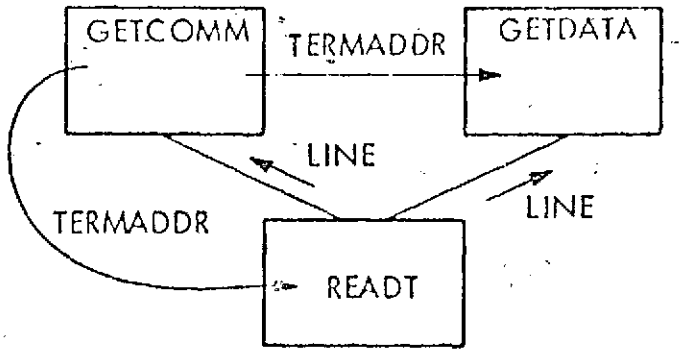
GETCOMM is a module whose function is getting the next command from a terminal. In performing this function, GETCOMM calls the module READT, whose function is to read a line from the terminal. READT requires the address of the terminal. It gets this via an externally declared data

element in GETCOMM, called TERMADDR. READT passes the line back to GETCOMM as an argument called LINE.

Note the arrow extending from inside GETCOMM to inside READT. An arrow of this type is the notation for externally-declared references.

So far, so good. Now, however, we wish to change this program. We need to create a module called GETDATA, whose function is to get the next data line (i.e., not a command) from a terminal. We recognize that it would be desirable to use module READT as a subroutine of GETDATA. These are at least five alternative designs, which are examined below.

1. GETDATA calls READT. Before it calls READT, it modifies TERMADDR in GETCOMM so that READT has the intended terminal address.



Note that we have probably created a bug. GETCOMM, as originally coded, never knew that any other module would change TERMADDR. Therefore, when GETCOMM executes after GETDATA, GETCOMM will be using the wrong terminal.

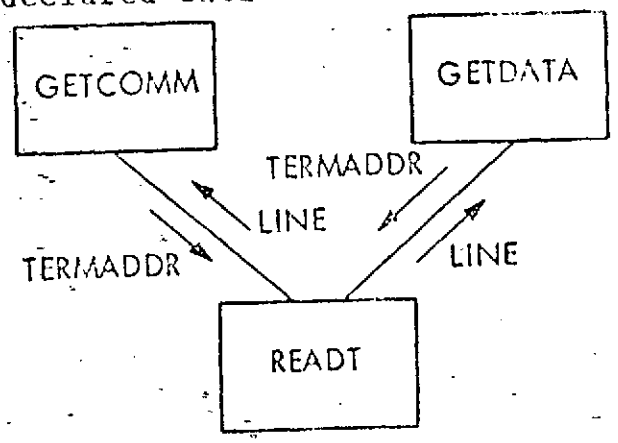
2. If the programmer of GETDATA recognized this problem, he might put instructions in GETDATA to save the current value of TERMADDR, set TERMADDR, call READT, and then restore the original value of TERMADDR. However, if there is a chance that GETDATA and GETCOMM can execute "simultaneously" (e.g., in a multiprogramming environment), then the bug still exists.

3. The programmer of GETDATA might recognize the problem and decide to modify GETCOMM. He changes GETCOMM so that is reinitializes TERMADDR each time it calls READT. This may eliminate the bug, but consider the cost. The programmer of GETDATA had to modify GETCOMM, a module which should have been independent of GETDATA.

4. The programmer of GETDATA might anticipate the above cases and decide that the easy way out is to code his own read line function, either within GETDATA or as another new module. This is unfortunate because he is

reinventing the wheel (by not using the existing READT module).

5. The programmer may recognize that the real problem is high coupling and may decide to clean it up. He makes TERMADDR an input argument to READT instead of an externally-declared data item.



In the long run, this alternative is best, but is also the most costly, since he had to modify both GETCOMM and READT.

This simple example shows that external coupling has an adverse effect on program modification, both in terms of cost and potential bugs. If GETCOMM and READT were not externally coupled from the beginning (i.e., TERMADDR was passed as an argument), the addition of GETDATA would have been much simpler.

A second type of external coupling is a reference to an externally-defined statement within a module, for instance, when one module branches to an externally-defined statement within another module. I leave it up to the reader to convince himself that this is at least as bad as the case of externally-defined data as shown above.

CONTROL COUPLING

Two modules are control coupled if one module passes elements of control as arguments to the other module. An "element of control" argument is an argument which directly influences the execution of the called module. Typical elements of control are function codes, flags, switches, etc.

Control coupling is undesirable because the two modules are not very independent. Since the calling module influences the execution of the called module (and, hence, has some knowledge of the internal processing of the called module), the called module is not a "black box". An additional side effect sometimes occurs with control coupling; many times the strength of the called module is low (i.e., not functional binding).

2. Data elements defined on the COMMON statement in FORTRAN modules.

3. A centrally located "control block" or set of control blocks (e.g., as in much of OS).

Common coupling causes the following weaknesses in the modules that are common coupled:

1. A modification of only several modules may impact every module that is common coupled to these modules. For instance, assume only two modules reference a data element X in the common environment. We desire to expand X from two bytes in length to four bytes.

   We make the necessary changes to the the two modules, but discover, to our dismay, that every module that references the common environment must be recompiled.

   In OS, most of the data elements are contained in system control blocks. These control blocks are mapped, element by element, in mapping macros. Any module which references a data element must contain the mapping macro of the proper control block. Anyone familiar with the ongoing development and maintenance of OS knows that the "macro problem" is a very costly problem. Since the mapping macros are constantly changing, and since it is infeasible to recompile the many thousands of modules whenever a macro changes, the modules always contain varying versions of the same macro. This has led to many bugs in OS and also to costly procedures to try to track and control this situation.

2. A desirable goal is limiting the references in each module to only those data elements which the module is supposed to reference. With common coupling, this is impossible, since each module can potentially reference every data element in the common environment. This leads to future problems in modification of these modules. For instance, when modifying a module, the programmer may decide to add a reference to another data element in the common environment. This can lead to bugs in instances such as (a) the other modules using this data element assume that they were the only users or (b) the programmer uses the data element for other than its intended purpose. In general, such modifications cause the data references in the program or system to become unstructured, uncontrolled, and often unknown.

   Again, we can use OS as an example. OS has an array called the Communications Vector Table in a well-known location in its memory. Almost every data element in the system can be located via the CVT. Hence, every module in OS is potentially common coupled. This has

led to costly problems in trying to keep track of, and control, which modules reference which data elements.

3. If a module references a common environment, it's very difficult to use that module elsewhere in the program or in another program.

   Assume, in a payroll program, we have a module named COMPFICA. COMPFICA computes an employee's F.I.C.A. deduction, using the salary as an input argument, but obtaining this year's F.I.C.A. rate from the common environment. We now desire to modify the payroll program, adding a function to compute, for the next year, the week when an employee's F.I.C.A. deductions will terminate. We desire to use COMPFICA with next year's F.I.C.A., but we face a possible problem, since if we temporarily modify the F.I.C.A. element in the common environment, we may cause a problem in some other part of the program.

   The use of a module that references a common environment in <u>another</u> program (i.e., a program <u>without</u> such a common environment) is very difficult. Generally, we have two alternatives, scraping the idea and writing a new module or creating a "fake" common environment before calling the module. The former is costly, since we are "reinventing the wheel." The latter leads to complex and cumbersome coding.

On the scale of coupling, common coupling usually sits between data coupling and control coupling. The exact placement of common coupling on the scale is dependent on its use. For instance, if control elements are placed in the common environment, the coupling is closer to control coupling. In this case, a combination of control and common coupling is worse than plain control coupling (e.g., passing control elements via parameters), so that control/common coupling is higher (worse) than plain control coupling.

On the other hand, we can see that the disadvantages of common coupling became less severe if the common environment is limited to a <u>subset</u> of the modules in a program. Hence, if common coupling cannot be avoided, it is desirable to limit access to the common environment to a minimal subset of modules. This tends to lower the overall coupling in the program. Furthermore, it is desirable to limit access to the common environment to the "top" modules in the structure, since this still removes the disadvantages of common coupling from the "lower" modules in the structure (e.g., allowing them to be used in other programs).

I have discussed common coupling in greater detail because its use is widespread today. Many people feel that common coupling leads to "generalized" designs. However, there is no objective proof of this and the only proof available leads

us to believe that common coupling leads to "ungeneralized" designs. Fortunately, the weaknesses of common coupling are beginning to be recognized. For instance, Belady and Lehman, in a paper on program maintenance and growth[1], made the following observations:

> These concepts reflect the accepted viewpoint that a well structured system, one in which communication is via passed parameters through defined interfaces, is likely to be more growable and require less effort to maintain than one making extensive use of global or shared variables.

DATA COUPLING

Two modules are data coupled if one calls the other and they aren't content, external, control, or common coupled. In other words, all input and output to and from the called module is passed as parameters or arguments. Also, all of the parameters are data elements (i.e., not control elements).

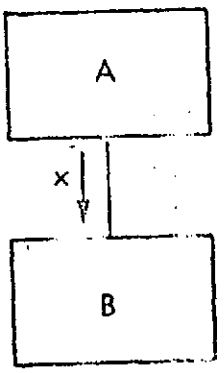
Data coupling is the lowest degree of coupling. Thus, modules that are data coupled are highly independent.

We can make a very strong statement about data coupling. Data coupling is a sufficient condition for any program. That is, any program can be written solely with data coupling. A proof of this statement is available.[4]

CONTROL VERSUS DATA COUPLING

In most cases, it is easy to distinguish between control coupling and data coupling by examining the parameters passed. However, for certain types of parameters, it is more difficult to distinguish between control information and data information. Several guidelines that may assist here are listed below.

1. The classification of the parameters (control and data) is dependent upon how the sending module perceives them, not how the receiving module perceives them.



If A passes x to B and A perceives x as data, then A and B are data coupled, even if B executes differently based on the value of x. If A perceives x as control information (i.e., A is telling B what to do), then A and B are control coupled.

The same argument applies to information returned from a module, such as return codes or error flags. If B passes a return code back to A saying "I've failed in performing my function" (implying that A can do whatever it wants), then this return code is data. If B passes a return code saying "I've failed; write error message XYZ", then A and B are control coupled because B is telling A what to do.

2. Control information is usually artificially created information, that is, information over and above the information being processed by the program.

   Suppose the function of B is to process a command. If A simply passes a command to B and B examines the command to determine how to process it, then A and B are data coupled. If A passes a command to B and, in addition, passes a code saying "process this XYZ command", then A and B are control coupled.

Point two also illustrates another disadvantage of control coupling. Control information is artificially created within the program and is extraneous. Hence, control information increases the complexity of the program because the program is dealing with extraneous and unnecessary data.

5. OTHER GUIDELINES

The following guidelines have an effect on the modularity of a program or system. They are used to guide the designer during the design process and also to improve a "first pass" modular design.

Principle of Parsimony

The Principle of Parsimony [5] (or "stinginess") means "never do more than you have to". It has two parts, simplicity and minimum commitment:

Everything else being equal, the simplest solution, design, interface, etc. is the best. This statement is very easy to prove and remember, yet it is often forgotten. Assume, everything else being equal, that we have two possible solutions, a simpler one and a more complex one. The simpler one, being easier to understand, has a more positive effect on the future maintenance and modification of the program.

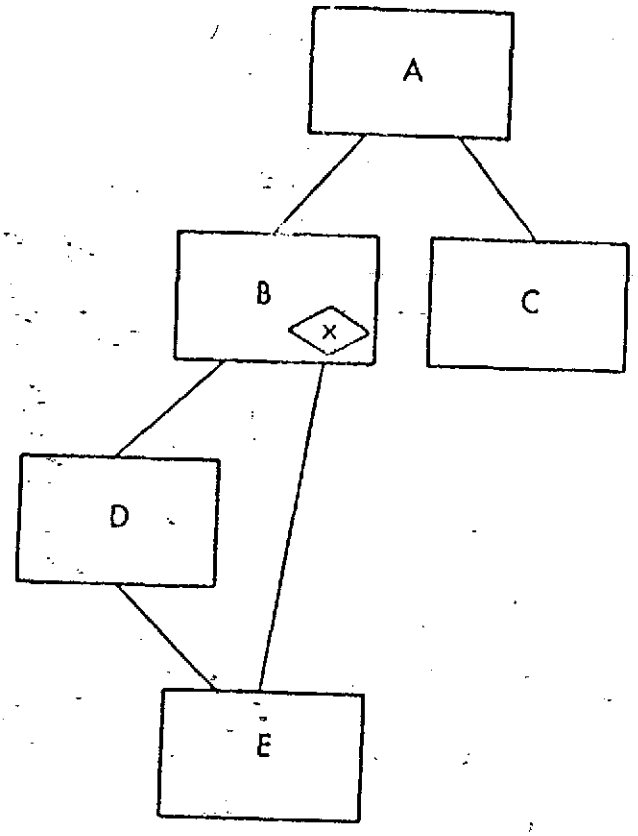
The idea of minimum commitment is that we should restrict the solution of a problem to solving no more than the immediate problem at hand. In other words, never design a program, module, interface, etc. to do more than it is required to do.

Many people have some unfounded ideas about designing a "generalized" program. They feel that generality has to be achieved via "open-ended" and "extendible" modules and interfaces, usually via extensive common areas or control block structures. In most cases, program designs of this type display low strength and high coupling!

The trap here is that we are poor prophets concerning the future modifications of a program. The best step we can take in producing a general and modifiable program is adhering to the guidelines of Composite Design and forgetting about all the misconceptions we may have.

Scope of Effect and Scope of Control

So far, I have encouraged thinking in structural terms, and discouraged thinking in procedural terms. However, life is not this simple. There are several relationships between structure and procedure worth understanding; one of these is the scope of effect and the scope of control[4].



The scope of control of a module is that module, plus all modules that are subordinate to that module. For example, the scope of control of A is A, B, C, D, and E. The scope of control of B is B, D, and E. The scope of control of C is C.

The scope of effect of a decision is the set of all modules whose execution is based upon the outcome of the decision. Assume module B contains a decision x. Decision x determines whether B will call D or E. The scope of effect of x is B (because different statements are executed in B depending on the outcome of x), D, and E. As a second example, assume decision x in B determines whether B should continue or whether B should immediately return to A and then have A call C. In this case, the scope of effect of x is A, B, C, D, and E.

The relationship between scope of control, a structural-concept, and scope of effect, a procedural concept, should be:

> The scope of effect of a decision should be a subset of the scope of control of the module containing the decision.

Examine the first scope of effect example above. The scope of effect of decision x was B, D, and E. The scope of control of module B (the module containing x) was B, D, and E. In this case, the rule isn't violated.

In the second example, we said that the scope of effect of x was A, B, C, D, and E. However, the scope of control of B is still B, D, and E, so the scope of effect is not a subset of the scope of control, and the rule is violated. Let's take a closer look at this violation of the rule and see what it implies.

We previously said that if x was "true", B would continue processing (e.g., calling D and E). If x was false, B would return to A and A would call C. Here's the problem! How does A know whether or not to call C?

Assuming that A and B aren't content coupled (e.g., B doesn't modify A), the common situation is for B to pass the results of decision x back to A. A then has to examine the result and decide whether to call C. Note that this decision in A is really a repeat of decision x in B!

We have already discovered two of the three problems that occur when the rule is violated. A and B are now control coupled, since B is passing control information to A. Hence violations of the rule usually lead to control coupling. Secondly, violations of the rule result in duplicate decisions being made in different modules. Lastly, violations of the rule weaken the strength of the modules. We can assume that decision x is part of the function performed by B. However, we had to repeat the decision in A. Perhaps this decision is not really relevant to the function performed by A. Hence, the strength of A may be lowered.

A technique for eliminating scope of effect - scope of control problems is discussed in section eight.

Module Size

Although there are no hard and fast rules for the size of a module, we can make some general statements about size.

You should take a close look at modules with less than 5 executable source statements or more than 100 source statements. Modules with a very small number of statements may not perform an entire function, hence, may not have functional binding. In addition, a system with a large number of very small modules may spend a disproportionate time in executing intermodule linkages. In some cases, these very small modules should be eliminated by placing their statements in the calling modules.

On the other hand, very large modules may be problem areas. Although the number of statements required to perform a function varies widely, there's a greater probability with a large-module that the module is actually performing more than one function. A second problem with large modules is understandability and readability. There is evidence to the

fact that a group of about 30 statements is the upper limit of what we can master when reading a module listing.[6]

In my experience, the "average" module contains between 10 to 40 high-level language executable statements. Also, there are usually a small number of perfectly valid very small modules and large modules. Size should not be taken as a firm rule, but it should be used as a signal to look for potential problems.

Recursion

Recursion occurs when a module is a subordinate of itself. That is, recursion occurs when a module calls itself, or when a module calls another module, which calls another module, etc., which calls the original module.

Programmers usually steer clear of recursion because they do not fully understand it. Also, the module linkage mechanisms of some operating systems and some programming languages do not support recursion.

The use of recursion in a modular design should be encouraged. Recursion tends to eliminate some redundant and excessive coding. To illustrate this, let's look at a simple example.

Assume we have the job of writing a module whose function is "write error message to terminal user, or, if this is unsuccessful, notify system operator." There are two arguments passed to this module, the error message and a terminal number.

Let's look at two alternate implementations of module WRITEMSG. The first alternative is to write the statements that write the message to the terminal, then check for an error and, if one occurs, write a message to the operator's terminal.

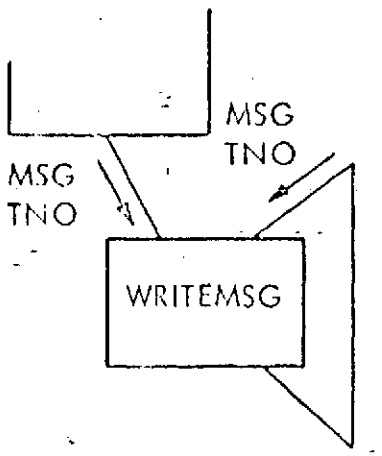
```
MODULE WRITEMSG (MSG,TNO)
          •
          •
          •
Set up and write msg to terminal
          •
          •
          •
IF NO ERROR, RETURN.
  ELSE
          •
          •
          •
Set up and write msg to operator's
     terminal

RETURN
```

Looking at this, we may detect some duplicate coding and decide, at the expense of added complexity, to share some of these statements between the "two halves" of WRITEMSG.

The other technique we could use is recursion. If the message can't be written to the indicated terminal, the module calls itself, passing an error message and the operator's terminal number as arguments.

```
MODULE WRITEMSG (MSG,TNO)
          •
          •
          •
set up and write msg to terminal
          •
          •
          •
IF NO ERROR, OR IF
TNO = SOTNO, RETURN.
     ELSE CALL WRITEMSG(EMSG,SOTNO)
     RETURN.
```

The structual notation for this is:



Predictable Modules

A predictable, or well behaved, module is one that, when given the identical inputs, operates identically each time it is called. Also, a well behaved module operates independently of its environment.

The most common violation of the first statement occurs when a module keeps track of its own state. The best example of this is a module containing a statement like "IS THIS THE FIRST TIME I'VE BEEN ENTERED? IF YES, THEN...." Modules of this sort are usually unusable in several places in a program, which violates one of the basic principles of Composite Design.

Consider a module called "GET NEXT INPUT TRANSACTION". Assume that this module, on its first execution, requests the operator to mount the required tape. Later, when modifying the program, we have a need in another part of the program for this same function (but using a different tape on a different tape drive). If we were to use this existing module, we may find our tape drive empty! To make matters worse, only one of the two tapes will get mounted, and this will depend on who calls this module first! Hence, the only way out is by spending more money either by writing a new module or by making this module predictable.

The second case of an unpredictable module is a module that makes assumptions about its environment, or, in particular, about its caller. As an example, a module was written to accept a string of messages as input, format them, and write them to a terminal. At first glance, this module appeared to be useful in several parts of the program. However, this proved to be false because of the implementation of the module. The programmer writing the module assumed that it would be called by only one other module. Therefore, before writing the string of messages, this module wrote an additional message, stating "ERROR IN FUNCTION XYZ. ERROR MESSAGES FOLLOW." Fortunately, this has a happy ending.

This mistake was discovered and the extra message removed. In this case, the calling module inserted this message into the string before calling the message module.

Part III THE DESIGN PROCESS

Parts I and II covered the "what" and "why" of Composite Design. Armed with part II, you could be relatively successful in leaving this paper and using the concepts of Composite Design. However, we have not yet really discussed the "how" of Composite Design. Part III describes the process of producing a modular design.

The steps in the Composite Design (and development) process are:

1. Starting with the problem statement (or functional specification, external specification), design the structure of the entire program or system using one or more forms of analysis.

2. Review the completed structural design, trying to maximize module strength and minimize coupling.

3. Review the design again, using the guidelines of section six (e.g., parsimony, scope of effect, size, recursion, predictable behavior).

4. Design the internal procedure (algorithm) of each module.

5. Repeat steps two and three.

6. Code the internal procedure of each module.

7. Proceed with the steps of unit (module) testing, integration, system testing, etc.

Step one is described in section seven. Step three is described in section eight. Steps four, six, and seven are covered in section nine.

COMPOSITE ANALYSIS

The design process, by definition, is a creative activity. Composite analysis, a technique for designing modular programs, does not replace creativity; it is meant to channel creativity in the right direction.

Composite analysis is a rough formalization of the design process. In this form, it will probably not strictly apply to any single design. The designer will have to adapt it, massage it, and compromise it to fit the particular problem he is trying to solve. In addition, two people independently designing the same program using composite analysis will probably arrive at two different modular structures.

The basis of composite analysis is that the structure of the program should resemble the structure of the problem. Hence, composite analysis involves an analysis of the problem structure and, in particular, the flow of data through the problem and the transformations that occur on that data.

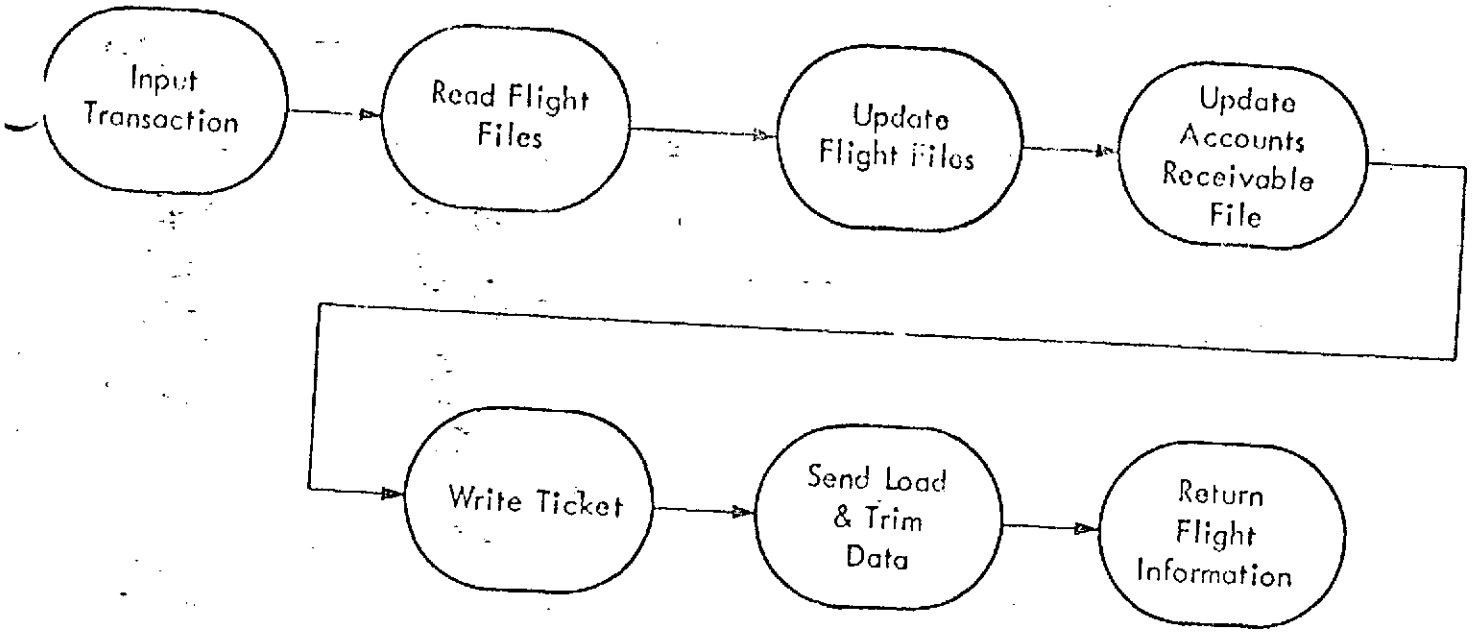
Note that in the preceding paragraph I stressed the word structure. Composite analysis is totally based on structure. When using it, do not think in terms of procedure, time, sequence, which event has to happen first, etc. In other words, think about what the program has to do; do not think about when the program has to do something.

I will describe the steps of composite analysis and then illustrate its use in an example. This example will then be refined in later sections.

Step One

The first step is sketching out a rough picture of the problem. Remember, this sketch should be in functional, not procedural, terms.

As an example, consider a simple airlines reservation system. It is driven by input from remote terminals. The major types of input are requests for information (e.g., seats available, flights), sales of tickets, and passenger check-in's. The rough structure of this problem is shown below:



Note that this diagram is non-procedural. For instance, on a request for seating information, only a few of the steps are performed. Also, for a ticket sale, we are not concerned about whether the ticket is written before or after the flight file is updated.

Step Two

Identify the external conceptual streams of data. An external stream of data is one that is external to the system. A conceptual stream of data is a stream of related data that is independent of any physical input-output device. For instance, we may have several conceptual streams coming from one input-output device or one stream coming from several input-output devices.

A good example here is OS. The input reader program may be reading from a physical input device (e.g., card reader). However, there are two conceptual streams here, the JCL statements and the "SYSIN" (i.e., those records following a DD * or DD DATA JCL statement). Since several input devices can be active simultaneously, the JCL stream is coming from several sources.

In our airlines reservation system, the external conceptual streams are the input transactions, tickets, flight information, and load and trim data.

Step Three

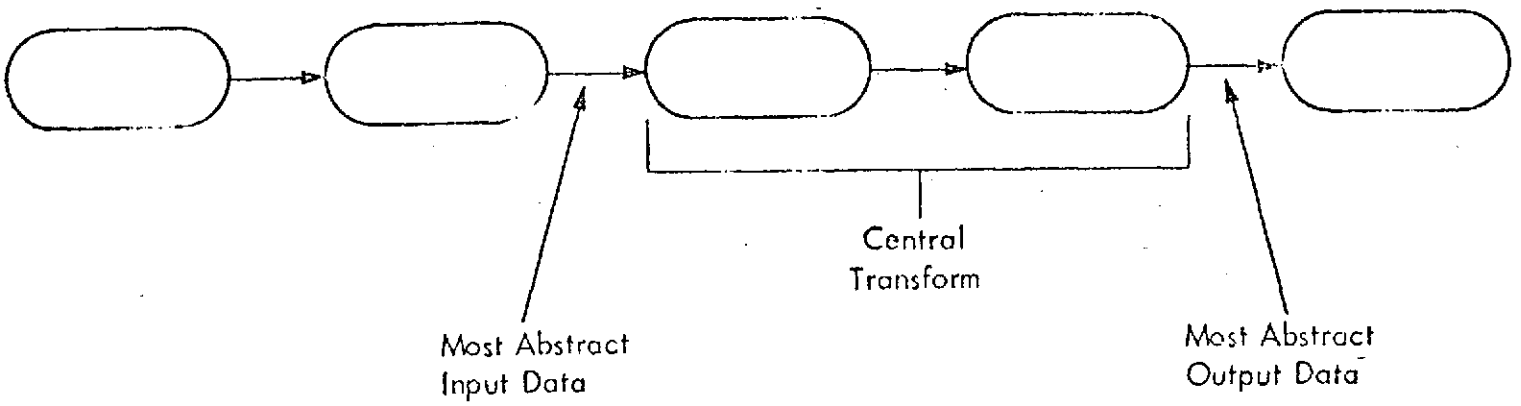
Identify the major external conceptual stream of data (both input and output) in the problem. Then, using the diagram of the problem structure, determine, for this stream, the points of "highest abstraction."

We're assuming that each problem will have a "major" stream of data. Any stream of data usually exists in many forms throughout the problem. For instance, in the airlines reservation system, the input transaction can exist in the following forms:

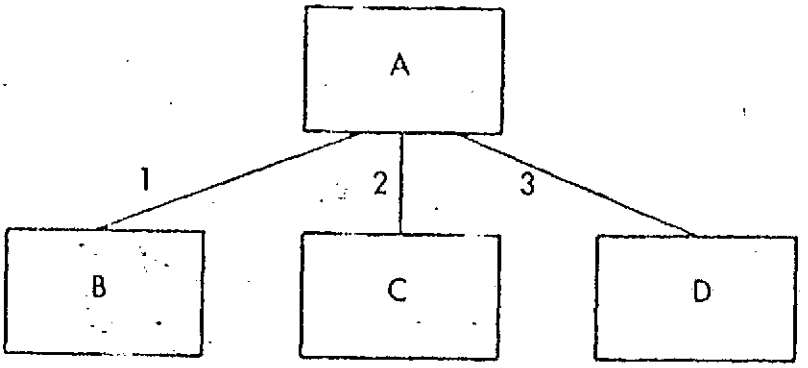
1. spoken words (from customer to clerk)
2. request typed into terminal
3. request received in digital form by computer
4. request formatted into meaningful internal format

The "point of highest abstraction" for a stream of data is the point in the problem structure where that data is farthest removed from its physical input or output form yet is still recognizable as being that particular stream of data. Hence, in the airlines reservation system, the most abstract form of the input transaction stream might be a validity checked input transaction in the proper internal format.

For the major input and output conceptual streams, we determine their points of highest abstraction. This defines two points on the problem structure. All information in the problem structure between these two points is called the central transform of the problem.



At this point, we begin to diagram the program structure.



	IN	OUT
1	Usually Nothing	Most Abstract Input Data
2	Most Abstract Input Data	Most Abstract Output Data
3	Most Abstract Output Data	Usually Nothing

The parameters passed are dependent on the problem, but the general pattern is shown above.

Step Four

The next step is to define the functions of the four modules developed in step three. Step four is the most important step in the process, since proper definition of these four modules is vital.

The function of each module should be described in a short, concise, and specific phrase. Remember, the function of a module is a description of the transformations that occur when that module is called. It does not necessarily describe the processing contained wholely within that particular module. With composite analysis, our objective is to define modules which have functional binding. In order to review some of the do's and don't's, it would be worthwhile to reread the section on module strength.

When module A is called, the program or system executes. Hence, the function of module A is equivalent to the problem being solved. If the problem is "write a FORTRAN compiler", then the function of module A is "compile FORTRAN program".

Referring to the diagram, we see that module B's output is the most abstract input data. Hence, module B should be defined as a functionally bound module whose function involves obtaining the major stream of data. An example of a "typical module B" is "get next valid source statement in Polish form." Because module B's function involves obtaining data, we refer to it as a source type module.

interfaces, or similar function) can be found, modify the modules to take advantage of higher fan-in to a common module.

5. When analyzing a subproblem, the source - transform - sink breakdown is more complex. The reason is that the major conceptual data stream of the subproblem normally enters or leaves through the module we are analyzing. For instance, if we're analyzing a source type module, the conceptual stream in this subproblem usually exits via this module (i.e., when it returns to its caller). Hence, this source type module may actually appear to be a sink type module with respect to this subproblem. The same applies to the analysis of a sink type module. A transform type module may act as both the source and sink with respect to its own subproblem.

This leads us to the following three guidelines.

6. The subordinates of a source type module are usually one or more source type modules and a transform type module. Occasionally, a source type module will have a sink type module as a subordinate.

7. The subordinates of a sink type module are usually one or more sink type modules and a transform type module. Only rarely does a sink type module have a source type module as a subordinate.

8. The subordinates of a transform type module are usually transform type modules. Also, some transform type modules will have sink and/or source type modules as subordinates.

Stopping

Step five is an iterative process. Yet, obviously, the process must eventually terminate.

There are no explicit criteria for stopping the composite analysis process. In practice, I've found that is "comes naturally". When none of the modules in the structure can be analyzed further into independent functional subordinate modules, then the composite analysis is complete.

An Example

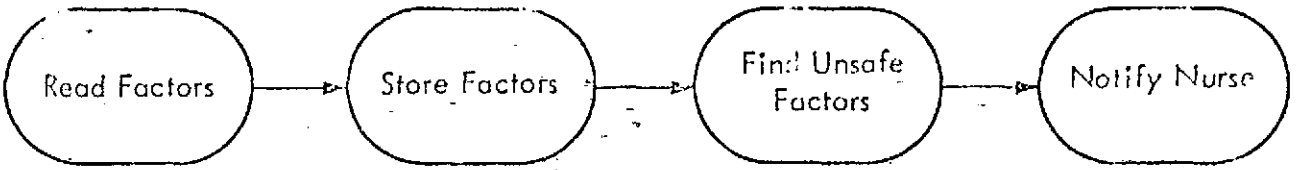
To better understand the use of composite analysis, we will use it in an example. Assume we have to design a program solving the following problem:

> Design a patient monitoring program for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program should read

these factors on a periodic basic (specified for each patient). The program stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified.

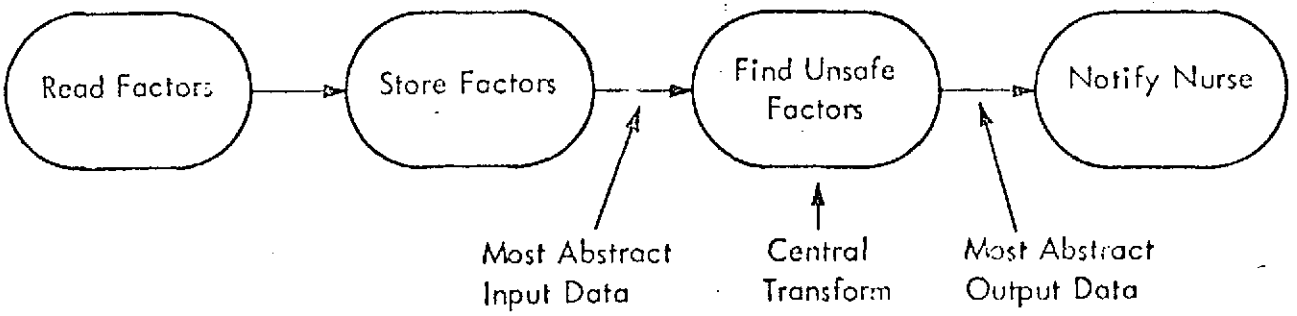
In a real-life case, the problem statement would contain much more detail. However, this one is of sufficient detail to allow us to design the structure of the program.

The first step is to outline the structure of the problem. This is shown below.

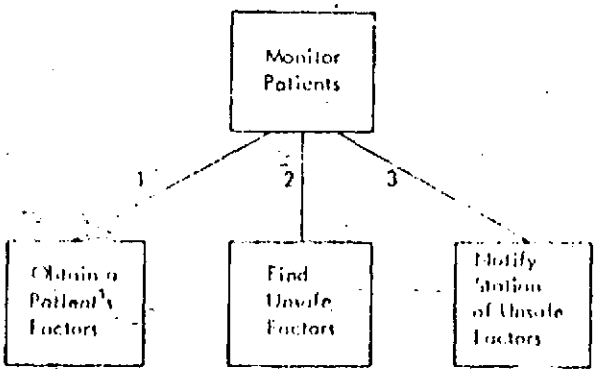


In step two, we identify the external conceptual streams of data. In this case, two streams are present, factors from the analog device and warnings to the nurse. These also represent the major input and output streams.

The point of highest abstraction of the input stream is the point at which a patient's factors are in a form to store in the data base. The point of highest abstraction of the output stream is a list of unsafe factors (if any).



We can now begin to design the program's structure.



	IN	OUT
1	Nothing	TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	List of Unsafe Factor Names and Values
3	PATIENTNUM and List of Unsafe Factor Names and Values	Nothing

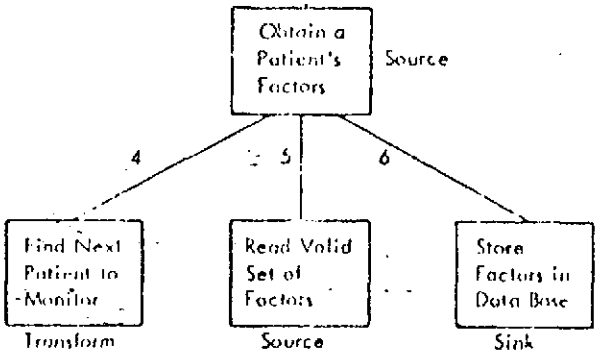
We will now analyze module "OBTAIN A PATIENT'S FACTORS". From the problem statement, we can deduce that this function has three parts:

1. determine which patient to monitor next (based on their specified periodic intervals)

2. read the analog device

3. record the factors in the data base

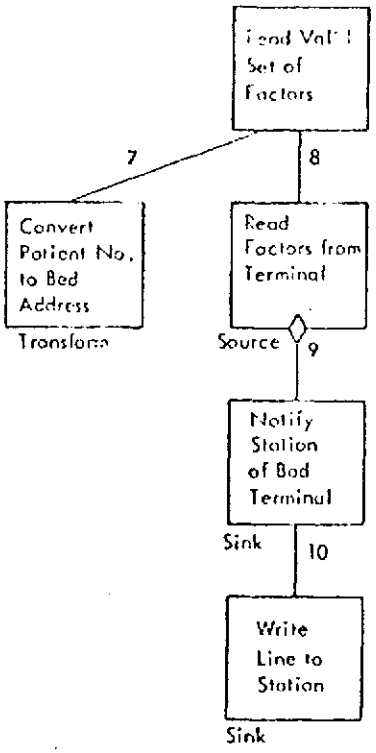
Hence, we arrive at:



	IN	OUT
4	Nothing	PATIENTNUM
5	PATIENTNUM	TEMP, PULSE, BP, SKINR, NOTVAL
6	PATIENTNUM, TEMP, PULSE, BP, SKINR	Nothing

Note that we have created transform, source, and sink type modules, respectively. NOTVAL is set if a valid set of factors wasn't available.

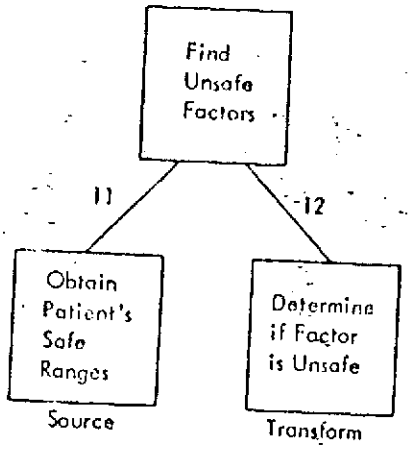
Further analysis of "READ VALID SET OF FACTORS" and its subordinates yields:



	IN	OUT
7	PATIENTNUM	BEDNUM
8	BEDNUM	TEMP, PULSE, BP, SKINR, NOTVAL
9	BEDNUM	Nothing
10	LINE	Nothing

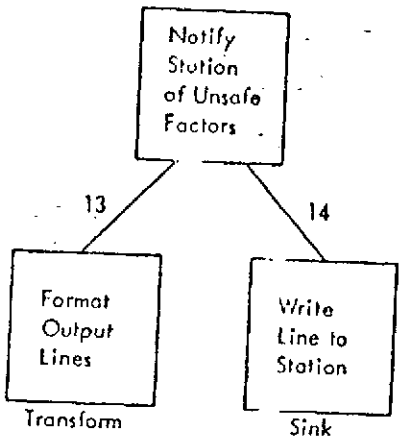
The diamond (decision) symbol indicates that the module is conditionally executed. That is, the nurses' station is notified only if the read from the analog device was unsuccessful.

Moving to another part of the structure, we analyze module "FIND UNSAFE-FACTORS" to arrive at:



	IN	OUT
11	PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
12	FACTOR, RANGE	UNSAFE

We now analyze that last part of the structure:

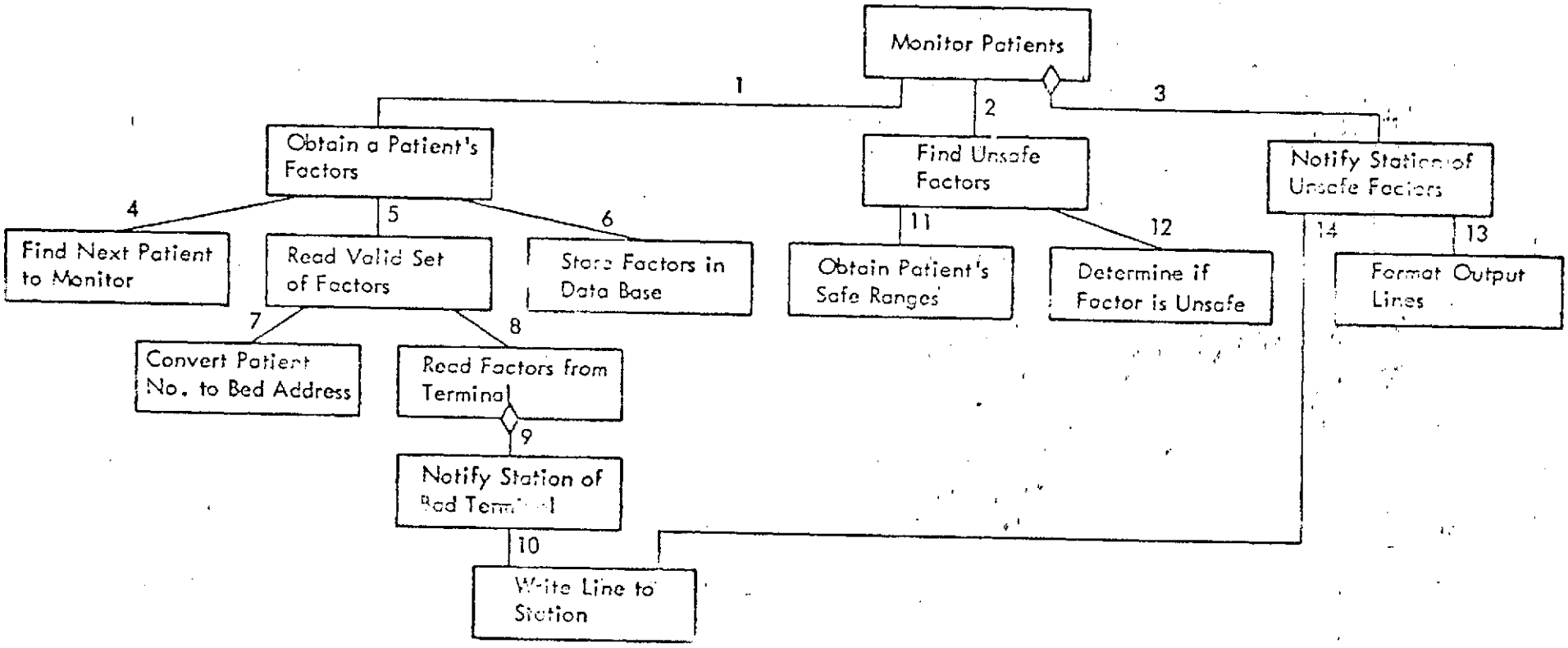


	IN	OUT
13	List of Unsafe Factor Names & Values	List of Lines
14	LINE	Nothing

The following sets of data are required in the program. These sets of data could either be passed down as parameters from module MONITOR PATIENTS or else be read from files.

1. list of patient numbers and their monitor time intervals
2. map of patient numbers to bed numbers
3. list of patient numbers and their safe ranges

The composite analysis of this program is now complete. The complete structure of the program is illustrated on the next page. Note that the design isn't complete yet. Several small flaws will be corrected in the next section.



	IN	OUT
1	———	TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	List of Unsafe Factor Names & Values
3	PATIENTNUM & List of Unsafe Factor Names & Values	———
4	———	PATIENTNUM
5	PATIENTNUM	TEMP, PULSE, BP, SKINR, NOTVAL
6	PATIENTNUM, TEMP, PULSE, BP, SKINR	———
7	PATIENTNUM	BEDNUM
8	BEDNUM	TEMP, PULSE, BP, SKINR, NOTVAL
9	BEDNUM	———
10, 14	LINE	———
11	PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
12	FACTOR, RANGE	UNSAFE
13	List of Unsafe Factor Names & Values	List of Lines

8. DECISION ANALYSIS

According to the overview in the introduction to part III, the steps following the analysis of the problem statement are:

2. Review the completed structural design, trying to maximize module strength and minimize coupling.

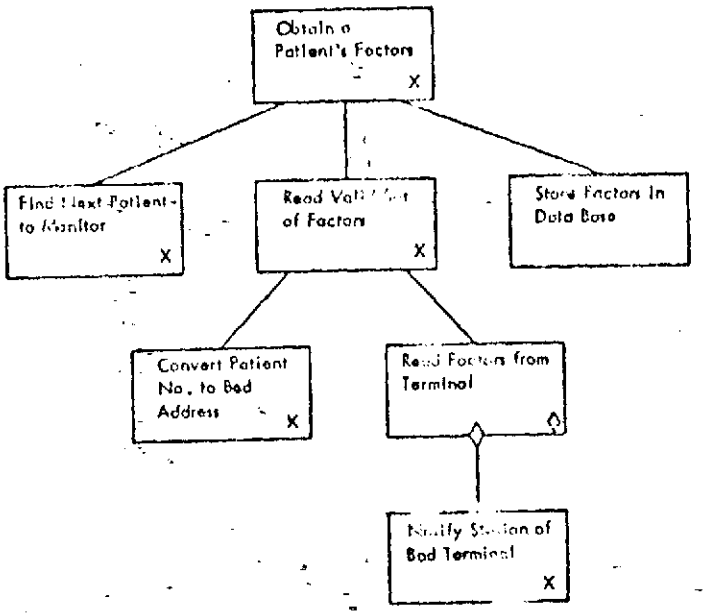
3. Review the design again, using the guidelines of section six.

These steps should be fairly straightforward, providing that you understand the concepts of part II. However, we will discuss one part of step three in more detail. That is, assuming you have discovered a scope of effect - scope of control problem, how do you solve it?

To review, the scope of control of a module is the set of modules consisting of that module and all subordinate modules. The scope of effect of a decision is the set of modules whose execution is based on the results of that decision. We said that the scope of effect should be a subset of, or equal to, the scope of control and, if this is not the case, we have duplicate decisions among modules and lower module strength.

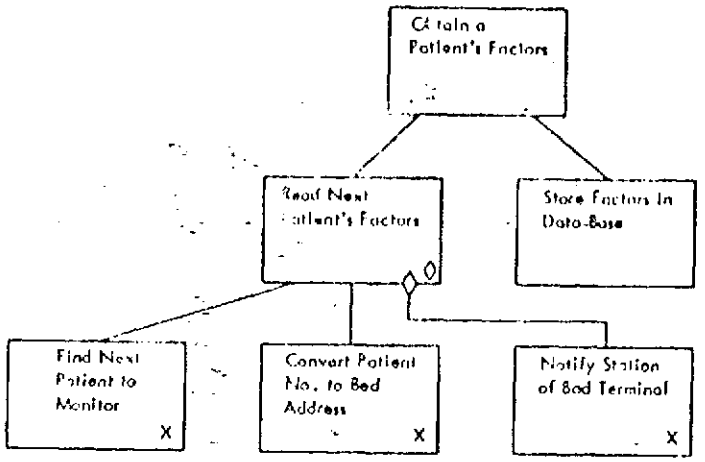
The cure for scope of effect problems is to redesign that affected part of the structure, either by moving the decision element "up" in the structure to where the scope of effect is no longer greater than the scope of control, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

To illustrate this, let's look at part of our patient monitoring program:



Note that module "READ FACTORS FROM TERMINAL" contains a decision asking "did we successfully read from the terminal?" If the read wasn't successful, we have to notify the nurse's station and then find the next patient to process.

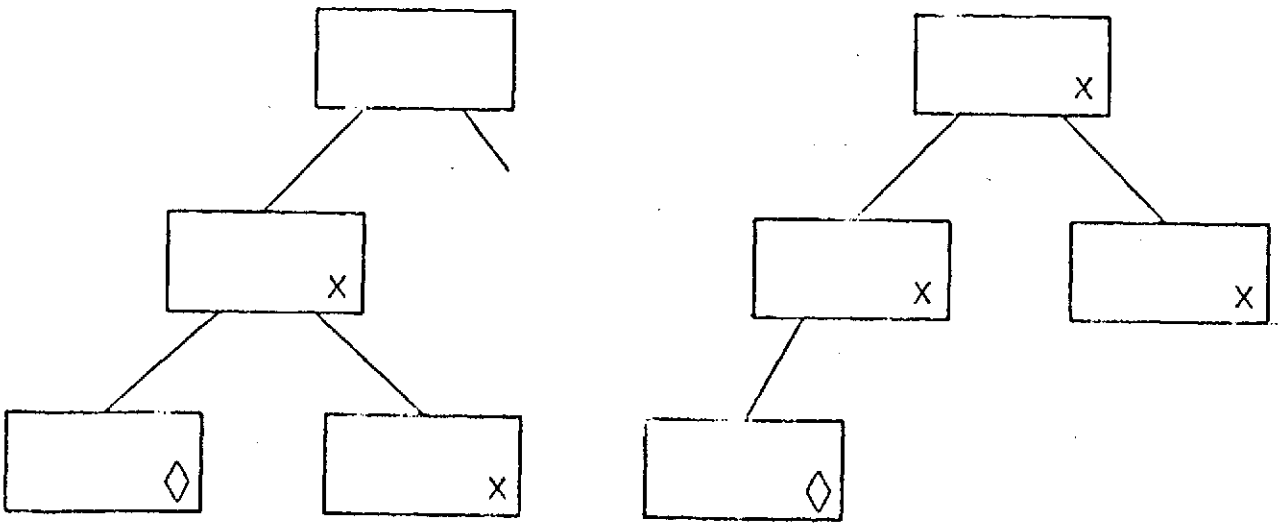
Modules in the scope of effect of this decision are marked with an X. Note that the scope of effect is <u>not</u> a subset of the scope of control. To correct this problem, we have to take two steps. First, we will move the decision up to "READ VALID SET OF FACTORS." We do this by merging "READ FACTORS FROM TERMINAL" into its calling module. We now make "FIND NEXT PATIENT TO MONITOR" a subordinate of "READ VALID SET OF FACTORS." Hence, we have:



Hence, by slightly altering the structure and the function of a few modules, we have completely eliminated the problem.

There are times when completely eliminating a scope of effect problem is infeasible or undesirable. In these cases, we try to minimize the problem, that is, minimize the difference between the scope of effect and control.

For instance, this: is better than this:



9. THE DEVELOPMENT PROCESS

After the structural design is complete, the next steps in the development of the program are the internal logic design of each module, a second review of the structural design, coding, and then testing. Although Composite Design is a design technology, a few observations about the development process (design, coding, and testing of each module) can be made.

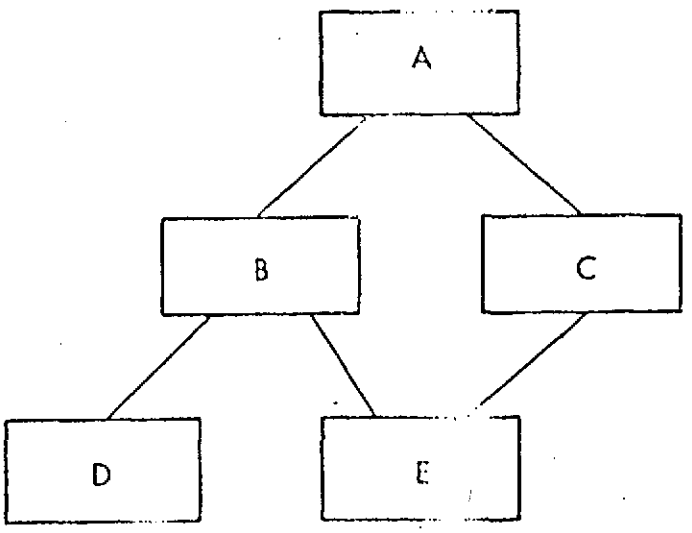
The most important consideration in program development is the requirement for a strategy and discipline. Almost any stategy or discipline is better than none at all (i.e., developing modules at random).

A second important consideration is that the design (structural and intra-module) should be completed before coding is started. Because the design process is an iterative process, coding should not start until the last design iteration has occurred. Because design is the most crucial and important phase of the project cycle, the design time should be lengthened and the coding time should e shortened.

Two development strategies that have evolved, "top-down development"[3] and "bottom-up development", warrant consideration.

Top-Down Development

In top-down development, coding is performed "top-down, in execution sequence". That is, the module at the top of the structure is coded first. Then, modules subordinate to this module are coded, this collection of modules is tested together, the next level is coded, and so on. However, the sequence is not quite this simple, as shown below.



Module A is coded first. In order to test module A, modules B and C are needed (or else "dummy" modules to simulate B and C). However, to test with B and C, D and E are needed. This presents us with a dilemma because it appears that the whole program must be coded before any part of it is tested. This dilemma is solved by developing top-down in execution sequence. Modules A and B are coded first (assuming A calls B before it calls C). Then A and B are partially tested (the part of A up to the call of C and the part of B up to the call of D). Then D is coded and the combination A, B, and D is partially tested. Then E is coded, A, B, D and E are tested, then C is coded, etc.

Note the following:

1. The complete testing of most modules is spread out over a long time period. For instance, A might not be completely tested until everything else is available.

2. The complete testing of the modules is not "top-down". For instance, D might be completely tested before A is completely tested.

3. The planning and control job can be complicated. For instance, knowledge of the execution of the program is necessary to determine the order in which modules are coded. Also, we have to keep track of how much of each module has been tested.

One definite advantage of top-down development is the resolution of module interfaces. If we code A before we code B, the parameters passed to B are well defined before we code B. If we code B before we code A, we may have to make assumptions about the parameters passed to B. In all cases, this is a real advantage of top-down development. However, the magnitude of the advantage is based on the quality of our original design. If the original design was good (e.g., interfaces were well defined), the coder will have to make a small number of assumptions concerning the interface.

A second advantage of top-down development is in the testing of modules. If we code and test B before A is coded, we may have to write a "dummy" module to call B in order to test B. However, depending on the testing environment and the nature of module B, this may not be a real advantage.

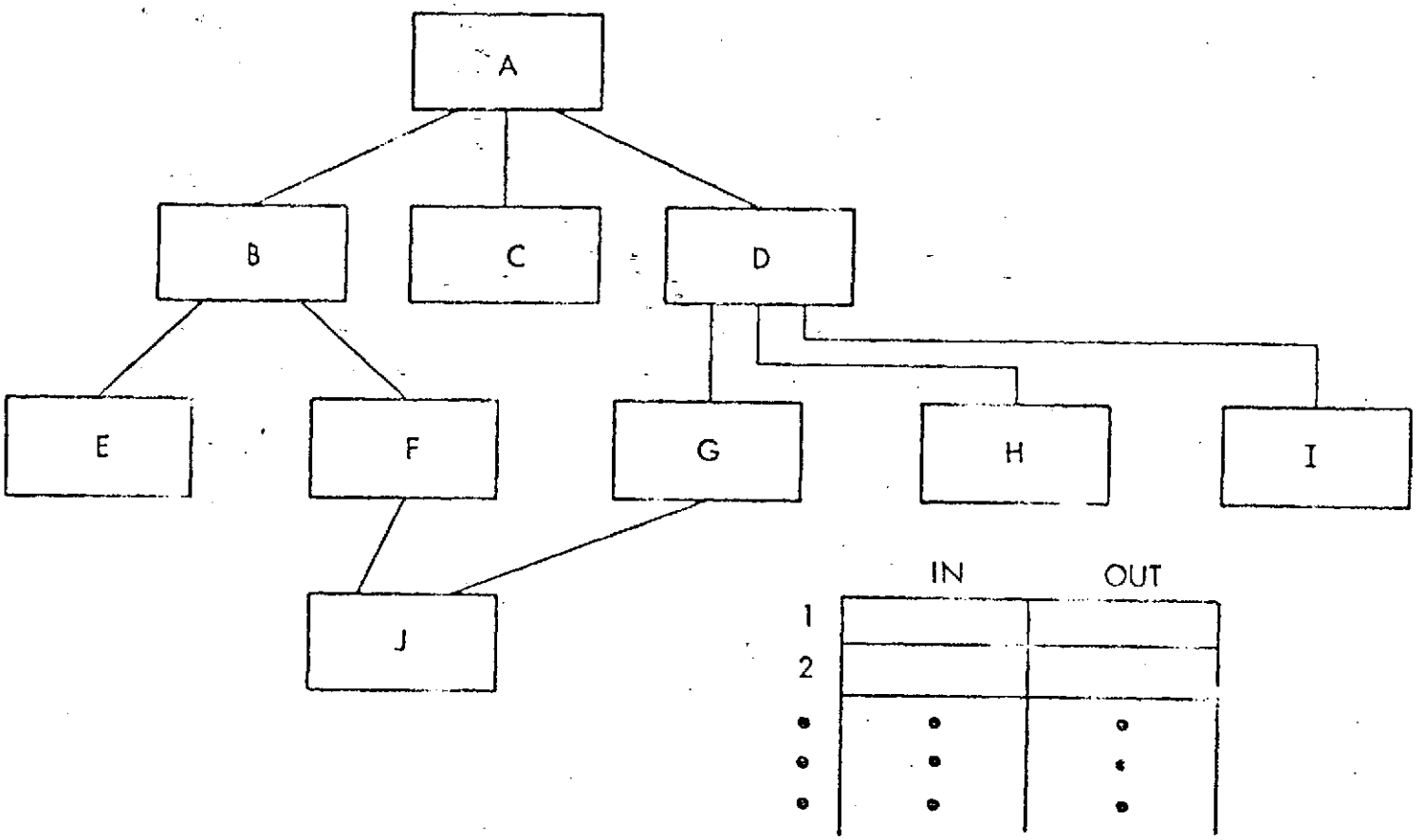
1. If an automated testing facility is used such as OS/VS2 TSO TEST or VM/370 CMS, the "dummy" module is unnecessary. These facilities allow us to describe the input parameters to module B and then execute B.

2. Module B may contain conditions that we wish to test which are very difficult to invoke through module A. Suppose module B contains code for "unlikely to occur" situations, such as validity checking the parameters

passed to it. If A always passes valid parameters, we cannot test this validity check in module B in a top-down fashion (e.g., we may still have to write a dummy driver to call module B, passing invalid parameters).

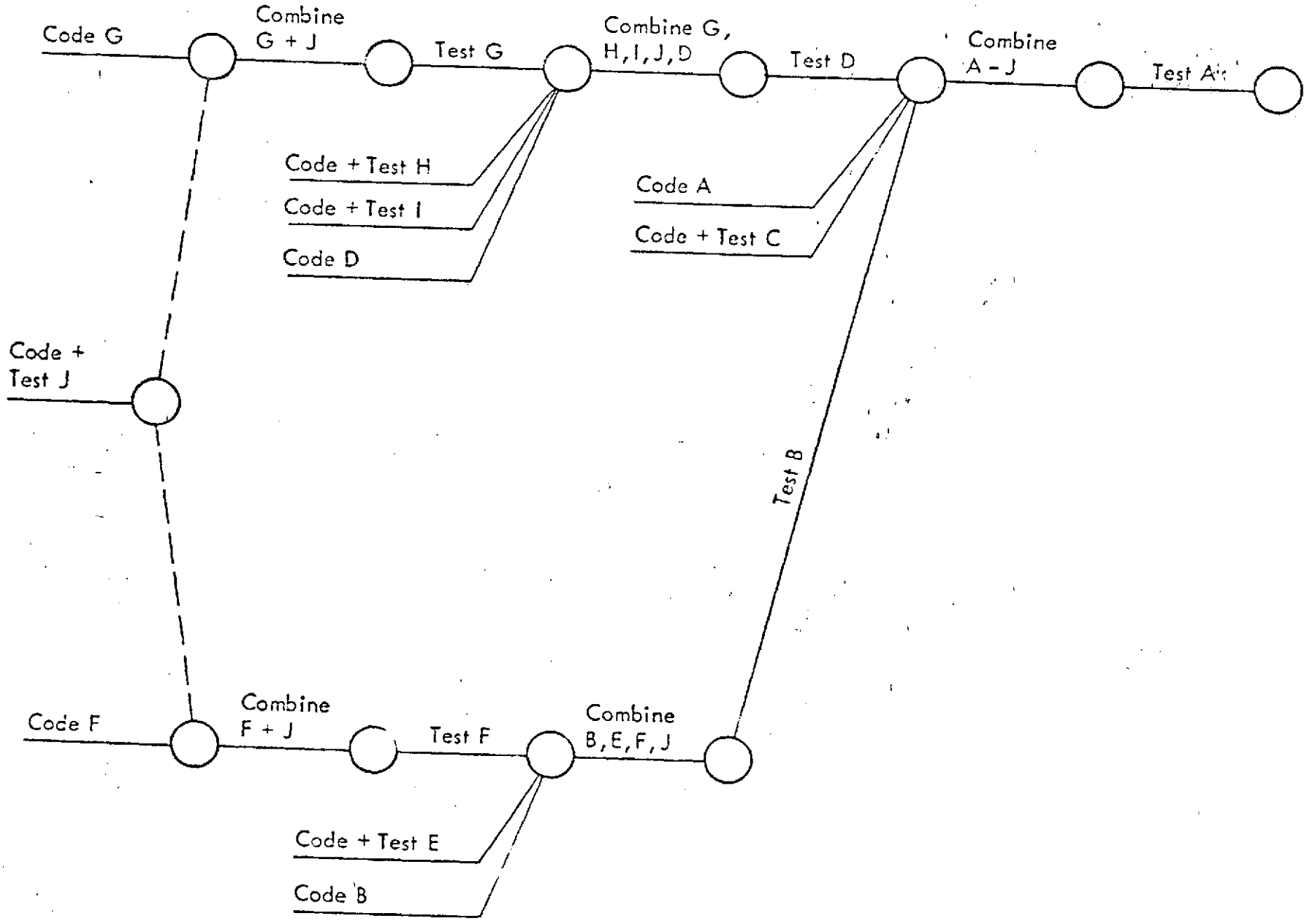
Bottom-Up Development

Like top-down development, bottom-up development involves a structured overlap of coding, integration, and testing. To illustrate this, consider the following example.



We start by developing the lowest modules (e.g., module J). Once J is developed and tested, we have extended the power of our "pseudo machine". That is, CALL J (x,y) is now in the statement repertoire, along with the standard statements such as A=B+C. Hence, we could now develop module F with the knowledge that CALL J is a working statement. To test F, we include the tested module J. We continue this integrating and testing process until we reach the top.

To illustrate this better, the following page contains a PERT chart showing the bottom-up implementation of the above program.


Code G
Combine G + J
Test G
Combine G, H, I, J, D
Test D
Combine A - J
Test A
Code + Test H
Code + Test I
Code D
Code A
Code + Test C
Code + Test J
Test B
Code F
Combine F + J
Test F
Combine B, E, F, J
Code + Test E
Code B

Note that this process almost eliminates the traditional unit test process. A unit test is a test of a single module in an isolated environment. However, in this process, the only modules that are strictly unit tested are the modules with no subordinates. Modules that have subordinates are not tested alone; they are tested with their subordinates.

In the previous example, module J was tested by itself (unit testing). However, module F was not tested by itself; it was tested by combining it with previously tested module J.

In addition, bottom-up development allows us to perform a large number of independent activities in parallel (refer to the PERT chart).

Which is Better?

You have probably noticed that I am somewhat biased toward bottom-up development. To review, the advantages of top-down development are a reduction in the assumptions made about interfaces and a potential reduction in the amount of "dummy driver" code written for testing. The advantages of bottom-up development are ease in planning and controlling the coding and testing processes, complete testing of every module at a single time, and the potential for performing a greater number of development activities in parallel.

Perhaps the conclusive argument can be determined by the presence of design changes to the program during its development. Since design changes are more expensive and usually more sloppy in parts of the program that are already coded, we should try to delay the coding of those parts of the program that are most susceptible to design changes. If the "bottom" levels of the program are most susceptible to design change, then top-down development may be advantageous. If we suspect that the "top" levels are most susceptible to change, then bottom-up development may be the answer.

The arguments for and against top-down development and bottom-up development aren't convincing. Either one can be used to develop a program designed with Composite Design. However, it is important to choose one or another and to stick with it.

Different programs have different susceptibilities to design change. For instance, we can point to a particular program and estimate that the majority of the design changes will probably occur in the modules toward the bottom of the structure. Unfortunately, I known of no general classifications that can be made to determine whether a program is more susceptible to change at the top or at the bottom. Perhaps until more research is done, the following guideline can be used:

If you estimate that design changes during the development of your program will primarily occur in modules toward the bottom of the structure, use top-down development. If you estimate that they will occur toward the top of the structure, use bottom-up development.

Part IV RELATED TOPICS

Part IV discusses several other aspects of Composite Design. Section ten discusses the management of a programming project incorporating Composite Design. Section eleven relates Composite Design to the virtual storage environment and discusses the physical packaging of modules in a paging environment.

Section twelve suggests several desirable attributes of future computer system (hardware and software) architecture to enhance the use of Composite Design. Section thirteen suggests some documentation standards to be used in describing the output of the structural design phase.

10. Project Management

Two of the biggest problems faced by programming project managers are a) resource imbalances, because programming resources (e.g., programmers) cannot easily be shifted around to match the current workload and b) inability to measure the progress of a project due to the lack of small measurable units. Composite Design can assist in the solution of these problems.

In a paper by Rhodes[7] the author states that the ideal "work unit" in a programming project should have the following characteristics:

finite logical function

defined start and end points

parameterization

fully testable

robust

small

We see that the "module" from our Composite Design concepts meets these criteria.

The output of a Composite Design activity is a structural diagram indicating the relationships among all modules, the function of each module, and definitions of all interfaces. If the design is good, the modules are robust (high strength) and very independent (low coupling). Because of these characteristics, programmer assignments can be easily shifted. Hence, we can shift programmers from module to module to smooth out the peaks and valleys in resource requirements. This gives us more flexibility in allocating manpower to meet changing conditions.

The fact that a modular program consists of many small modules also enhances this "smoothing" capability. Furthermore, since we start with a large number of small work units (modules), more precise planning of programmer workloads is possible.

The second problem I mentioned was our inability to accurately measure the progress of a project. This problem is usually caused by too few points of measurement and ambiguously defined points of measurement. Envision a program consisting of one large module. Checkpoints such as 50 percent code written or 50 percent test cases successful are usually meaningless. First, their meanings are interpreted differently by different people. Secondly,

they're misleading, since 50 percent code written does not imply that the coding effort is half complete.

The answer to this problem is measuring progress based on a number of smaller activities (e.g., implementation of modules in a modular program).

By using one of the development processes suggested in section nine, we see that the development schedules and assignments can be directly related to the structure of the program (see the PERT chart in section nine). Hence, the structural diagram produced by the system designers using Composite Design gives the project manager a basic development plan!

Estimating

Another problem facing project managers is arriving at good estimates of the resources required. A modular program design makes estimating easier because it is easier to predict the size of each module or to predict an "average" module size (see reference 8).

New Programming Step

When Composite Design is to be used on a project, its use should be explicitly recognized by creating a step in the programming process for it. Hence, a step called structural design should be identified in the project plan. It should follow external specification design and precede the logic design of each individual module. Documentation produced by this step is discussed in section 13.

Other Hints

Once a good modular design for a program has been developed, it would be unfortunate to degrade this design when the program is modified in the future. The project manager should insure that all future modifications to the program adhere to the principles of Composite Design.

The same applies to program maintenance. When a bug is found, there is a temptation to find a "quick and dirty" fix. This temptation should be fought, since a "quick and dirty" fix may fix the current problem, but it may also degrade the design of the program, resulting in higher future maintenance and modification costs.

By following one of the development processes described in section nine, we see that the implementation plan (internal module design, coding, and testing) can be directly derived from the structural design. By constructing a PERT diagram from the structural design, we have the basic implementation plan. Bu matching the PERT chart with the resources (e.g., programmers) avaialble, individual programmer assignments and

schedules can be derived. Note that specific individual assignments for the implementation phase cannot (and should not) be made until the modular design is complete.

11. Modularity and Virtual Storage

The concept of virtual storage is heralded as a mechanism to make the programmer's job easier, since it lessens the programmer's problems of designing for a particular memory size, packaging programs into overlays, etc. These claims are certainly valid. However, to insure adequate performance, the programmer must now worry about the effects of paging on his program.

Performance in a paging environment is inversely related to the number of page faults incurred. A page fault is the interruption that occurs when a reference is made to a page which is not currently in real memory. Hence, the performance minded programmer should attempt to find an optimal packaging of his program and data, that is, a packaging which minimizes the number of page faults. Experiments in this area have shown that proper packaging has resulted in a five-to-one reduction in the number of page faults.

Throughout this paper, I have repeatedly warned against thinking about the procedural aspects of the program. This warning must be dropped for this section, because packaging of a program in a paging environment is entirely a procedural problem. That is, it is based on the execution characteristics of the program.

To package a program in a paging environment, we need the following information:

1. the size of a page (I'm assuming the system has a single fixed page size).

2. the size of each of the modules in the program.

3. a structural diagram of the program.

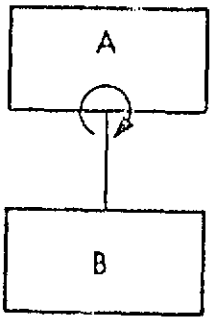
4. some knowledge of the procedural aspects of the program, in particular, the "when" and "why" behind the calls to each module.

The process we are discussing involves the proper physical placement of modules among pages within the virtual storage to minimize page faults. Since packaging is a procedural problem, and since it requires the output of the design phase, packaging cannot, and should not, be considered during the design phase. Because it requires knowledge of each module's physical size, packaging cannot normally be considered until after the coding phase.

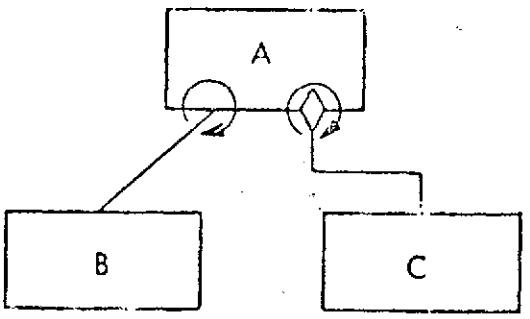
Packaging is primarily an art. I will list several prioritized guidelines for packaging and then illustrate their use in an example.

Priority One - Iterations

Group together modules which call one another interatively. For instance, if module A iteratively calls module B, then A and B should be grouped together.

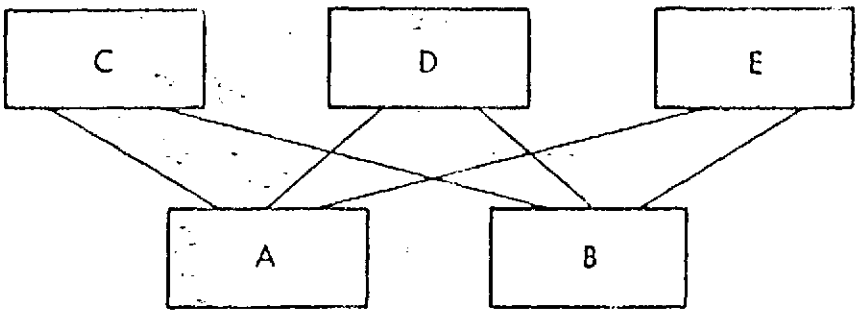


Probability of execution is another factor. If A repeatedly calls B and C, and it repeatedly calls B every time A is entered but repeatedly calls C only sometimes, then A grouped with B is of top priority, and A grouped with C is of lesser priority.



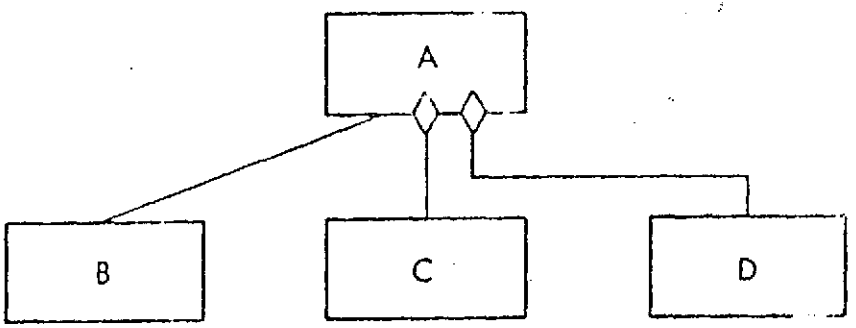
Priority Two - High Fan-in

Groups of modules with a high fan-in (number of calling modules) that are called by the same set of modules should be grouped together. For instance, if A and B are called by a set of modules (C, D, and E), then A and B should be grouped together.



Priority Three - Frequency

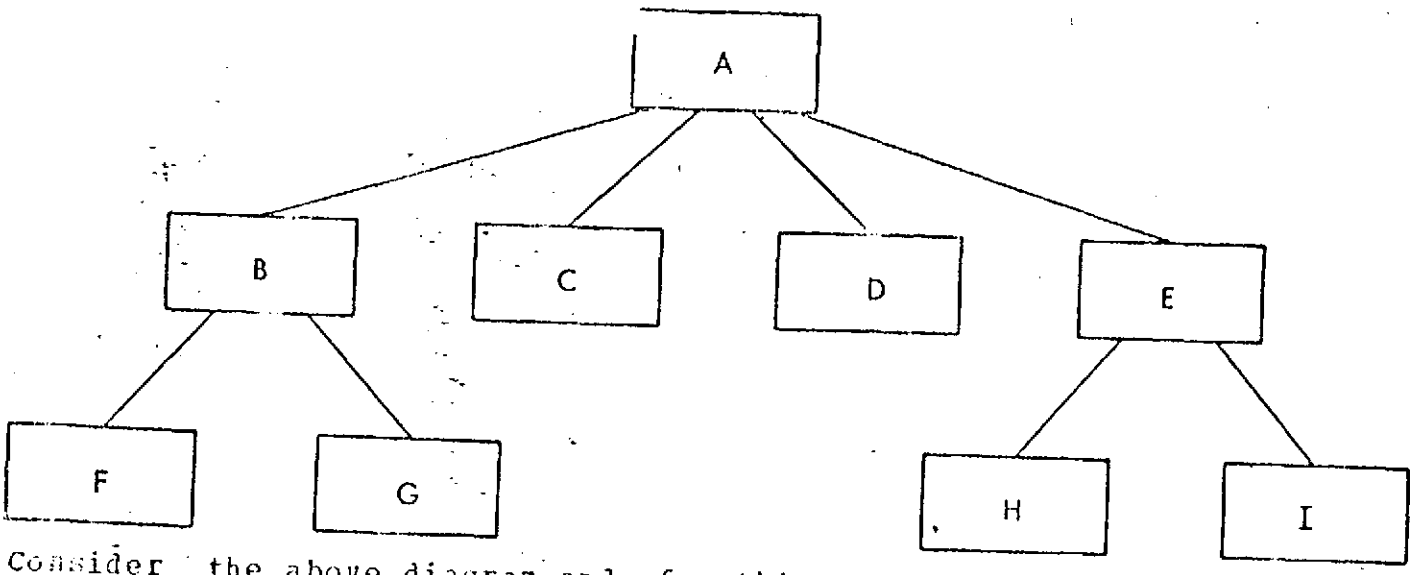
Group together those modules which call one another most frequently. For instance, in the following diagram, A calls B every time A is executed, A calls C about fifty percent of the time, and A calls D very infrequently.



Our first concern should be to group A and B together. Grouping A and C (and B) together is of lesser importance, but desirable if possible. We probably should avoid grouping A and D together.

Priority Four - Execution Sequence

If the above three priorities have been exhausted and space still exists within the pages to add additional modules, execution sequence should be the next consideration. That is, modules which will execute sequentially should be grouped together as much as possible.



Consider the above diagram and, for this example, assume the execution sequence is:

A B F B G B A C A D A E H E I E A

Assume that we can fit any five of these modules on one page and the other four on a second page.

By scanning the execution sequence and picking any sequence of five unique names, we have the optimum solution. For instance, if we pick the first five unique names in the sequence, we have A, B, F, G, and C in the first page and D, E, H, and I in the second page. This yields four page faults in the execution (two in the call of D from A and the return to A and two in the call of E from A and the return to A).

In fact, we can pick any five modules together in the sequence. For instance, starting at position four, we could pick B, G, A, C, and D for one page, which still yields four page faults.

If we do this <u>without</u> considering the execution sequence, the page faults are greater. For instance, putting A, B, C, F, and H in one page yields eight page faults. Putting A, B, C, D, and E in one page also yields eight page faults.

Even when the execution sequence cannot be determined (e.g., decisions in the modules alter it), this technique can be used by determining the <u>most probable</u> execution sequence.

<u>Non-Grouping Criteria</u>

In addition to the above prioritized guidelines for grouping modules on pages, there are several cases where we can identify modules that <u>shouldn't</u> be grouped together. Not grouping modules of this sort together will allow us more freedom to make additional desirable groupings.

Any module that is only executed once should be separated from the other modules. Also, any modules that provide infrequently used optional functions should be separated from the other modules.

Example

The diagram on the next page will be used as an example. The module sizes are indicated in the lower right hand corners of each module. Assume, in this system, that the page size is 1000 bytes.

The first step is to find priority one groupings. Examining the iterations, we arrive at the following three groups:

(B, D, E, F, G, H) (D, L) (C, I, J, K)

The next step is to find priority two groups. Modules R, S, T, and U meet the criteria, since they are called by a common set of modules. Hence, the single priority two group is (R, S, T, U).

Priority three groups are now developed. Making a few assumptions about the program, we determine that there are two groups of high frequency calls. They are (R, S, T, U, N, O, P) and (P, J).

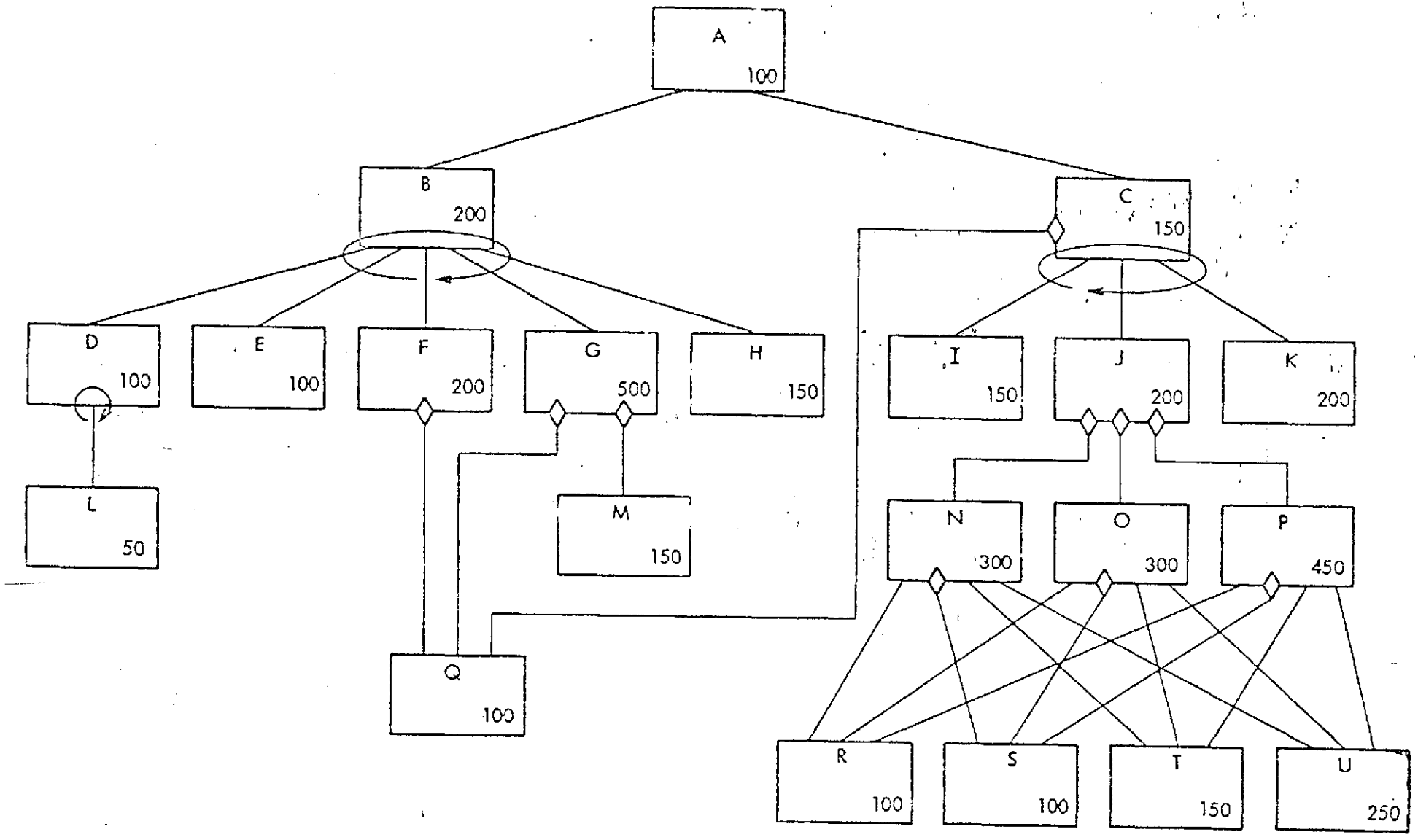
The fourth priority is execution sequence. Rather than determine this now, we will omit it and come back to it later only if the first three priorities are insufficient.

Looking at the first priority one group, (B, D, E, F, G, H), we should start by packaging these modules together. However, their combined size is 1250 bytes. We have to omit something so we place B, D, E, F, and H (750 bytes) on page 1 and G on page 2. The next group is (D, L). Since D is on page 1, we can include L on page 1 for a total of 800 bytes on page 1.

The next priority one group is (C, I, J, K). Since they cannot fit on pages 1 or 2, we put them on page 3 (combined size of 700 bytes).

The only priority two group is (R, S, T, U). Since this cannot fit on the first three pages, we place it on page 4 (now 600 bytes).

There are three priority three groups, (R, S, T, U, N, O, P) and (P, J). The first group has a size of 1650, so it can't be placed intact on page 4. Since the other group also contains P, we'll put P and J on a new page, page 5. The (R, S, T, U, N, O) group is still too large (1200 bytes) so we must remove N or O. If we remove O and put it on page 5, page 4 now contains 900 bytes and page 5 contains 950 bytes.


A
100
B
200
C
150
D
100
E
100
F
200
G
500
H
150
I
150
J
200
K
200
L
50
M
150
N
300
O
300
P
450
Q
100
R
100
S
100
T
150
U
250

The two remaining modules are M and Q. Page 2 currently contains module G, and Q and M are connected to G by execution sequence, so we put M and Q in page 2.

The packaging is:

Page	Modules
1	B, D, E, F, H, L
2	G, M, Q
3	C, I, K, J
4	R, S, T, U, N
5	O, P, J

Experimentation

Now that we have determined a way of packaging a particular program, further optimization is probably worthwhile. The following procedure is suggested:

1. Pick one or more "most probable" executions of the program.

2. For these cases, write down the execution sequences by module.

3. Make an assumption about the number of page frames in real storage available to the program. That is, assume the program will always have X pages in real storage.

4. Walk each of the execution sequences from step two through the modules. On paper (and in your mind), perform the paging and count the number of page faults. You can either use the paging strategy of the system that this program will execute on or assume a paging strategy (e.g., demand paging with replacement of the "least recently used" page).

5. Now, make a change to the packaging. For instance, if there were some arbitrary decisions made in the original packaging, you can change these decisions. Repeat step four and compare the results (number of page faults).

It should be obvious that this section has described an art, not an exact science. For a more sohpisticated technique involving the examination of the reference patterns of a program, see the paper by Hatfield and Gerald. [9]

12. Agreeable Hardware and Software

In today's environment, there are obstacles to modular programs. These obstacles are in the form of CPU architecture and software design (operating systems and programming languages).

The linkage editor is one of the biggest enemies of modularity. Most operating systems have a program called a linkage editor. The linkage editor combines a group of modules into a single named entity called the load module. Hence, the linkage editor is a "demodularizer."

Once a module is link edited within a load module, the module loses its identify outside of the load module. Hence, if module A is in load module M, then we cannot call module A from a module that is outside of M. This restricts one of the basic concepts of Composite Design -- that a module should be accessible and usable by a large number of other modules.

The linkage editor should be eliminated, or, if it is used, we should have only one module per load module. All linkages between modules must be dynamic (e.g., LINK facility in OS). The host operating system must provide a fast dynamic linkage facility.

Another function of the linkage editor is to resolve external symbol references among modules (external coupling). If we eliminate the linkage editor, we also lessen the problem of external coupling (i.e., the programmer is forced to use another, hopefully better, form of coupling).

The operating system and programming languages must enforce a high degree of data isolation among modules. This includes such things as allowing no module to modify another module and making names defined within one module local to that module. Most programming languages do fairly well here but APL is an exception. APL has a weakness in that names in a module (function) are global unless explicitly declared as local. Hence, APL promotes common coupling.

Standard linkage conventions must be followed for all language processors to allow modules written in different languages to call one another.

Recursive modules must be supported.

Operating systems and programming languages must further isolate modules by restricting a module's external reference to only those items explicitly declared as input and output. For instance, most languages give too much freedom to modules in dealing with their arguments. In the following sequence

```
CALL M (A, B, C)
```

module M can modify A, B, and C, even if A and B are intended only as inputs and C is the only intended output. Programming languages should support the following:

```
CALL M IN (A,B) OUT (C)
```

Operating systems must provide the necessary storage protection for this, that is, so that the only external data module M can read are A and B and the only external data module M can modify is C.

An argument that is sometimes valid is one concerning efficiency. If intermodule transfers are slow, then a high degree of modularity may result in a large amount of overhead in just transferring from module to module. The four elements of a module call are:

1. transmission of arguments
2. transmission of the return address
3. saving of the calling module's state (e.g., general registers)
4. allocating of private storage for local variables

In OS, a dynamic module linkage between two reentrant modules takes the following steps:

1. an address vector of the arguments is built in a temporary storage area.
2. CALL is issued. For modules in the same load module, a direct branch is used. For other cases, the LINK function is used.
3. The general registers are saved in a temporary storage area.
4. A temporary storage area is obtained (via the GETMAIN function) for local storage and save areas.

Of these, step four causes the most overhead. Storage must be allocated during every CALL and freed during every return. The GETMAIN and FREEMAIN functions involve a significant number of machine instructions, in fact, normally more than the instructions executed in the module itself! Operating systems must improve significantly in this area in order to better support modular programs.

A second area requiring further imporvements is step two, the LINK facility.

Possibly the best answer to improving module linkage is hardware assisted linkage. If increased program modularity

3. Documentation

The output of the structural design phase should be a) a description of the structure of the program (i.e., who calls who), b) a description of the inter-module interfaces, and c) a description of each module (i.e., its inputs, outputs, name, and function). Earlier sections sufficiently described a and b, and appendix A defines the notation. This section discusses point C, a description of each module.

This section will not discuss the various techniques (natural language, specification language, graphical, etc.) that can be used to describe a module. Instead, it discusses the types of information that should be contained in the specification.

A module should have two types of specifications, an external module specification which describes only that information needed by a module that calls this module, and an internal module specification which describes the internal logic (operation) of the module. It is important to distinguish between, and physically separate, these two specifications because the internal specification can be altered without affecting the calling module but changes to the external specification usually require changes to the calling module.

The internal module specification is written during the implementation (development) phase. The external module specification is written earlier, near the end of the structural design phase. In this section, we will only discuss the external module specification.

The external module specification should describe all the information needed by the calling module, and nothing more. Hence, this specification should describe the module's name, inputs, outputs, and function.

Module Name

This is a description of the name that is used (e.g., in the CALL statement) to reference the module. Module names should be descriptive of the function performed by the module.

Function

The function performed by the module should be described in a single sentence and then with an expanded description, if necessary. The expanded description could be a narrative description, decision table, graphs, etc. Note that only the module's function, not its internal logic or operation, should be described here.

Inputs

This is a precise description of all input data to the module. This should include a description of all input parameters, their physical order, their format (e.g., size and type (binary, decimal, character)), and the range of valid values.

If the module is other than data coupled, input descriptions will be more complex.

Outputs

This is a precise description of all output data from the module. This includes output parameters, their physical order, format, range, and error information (e.g., return codes). If different classes of output may be returned, then the output should be described in terms of "cause and effect" relationships with the input. Again, if the module is other than data coupled, output descriptions will be more complex.

Often, a module's specifications are contained at the beginning of the module in a "module prologue", a group of standardized comment statements. When this occurs, the prologue should not indicate which modules call this module. If the prologue of module B states that it is called by module A and we later add a module C which is to call module B, we have to alter module B (update its prologue), which conflicts with the goals of modularity.

Note, however, that although a module's specifications should not reference the calling modules, a module's internal specifications will normally describe any calls to other modules from this module. Hence, module specifications should only describe processing in that module and any subordinate modules; they should make no reference to any other module.

ACKNOWLEDGEMENTS

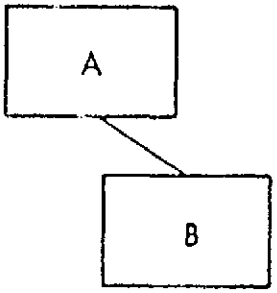
The ideas on scope of control and scope of effect and several of the module strength and coupling measures are due to Mr. Larry Constantine of the Information and Sciences Institute.

I am also indebted to Mr. Don Hooton and Mr. Ralph Childers of IBM's System Products Division for several discussions that helped clarify some of the points in this paper.

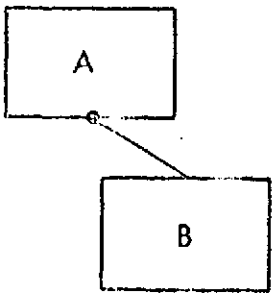
APPENDIX - NOTATION

Module

Predefined module
(e.g., pre-existing module).

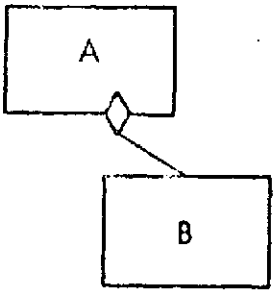


Module A contains a call to module B. Module B is subordinate to module A.

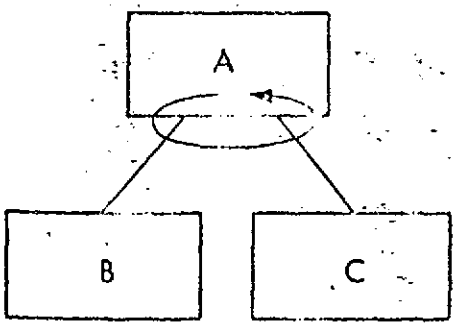


Module A transfers control (without return) to module B.

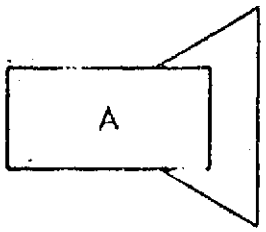
For example, XCTL in OS.
Not recommended.



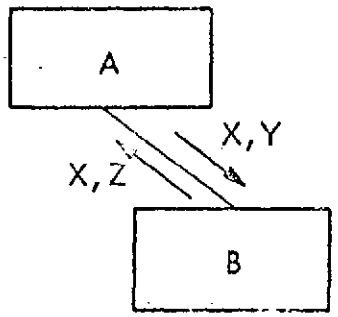
Conditional call. Module A sometimes (not always) calls module B.



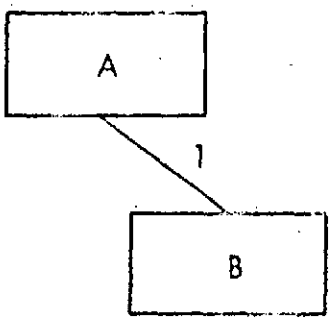
Repetitive call. Module A iterates through calls to B and C.



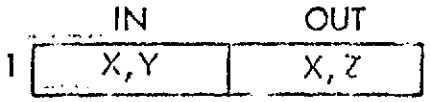
Recursive call.
Module A calls itself.



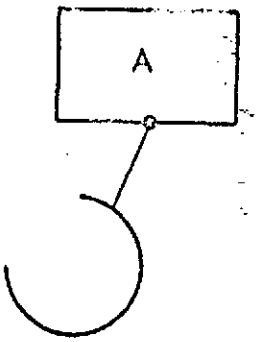
Module A calls module B passing parameters X, Y, and Z. X and Y are input to B; X and Z are output from B.



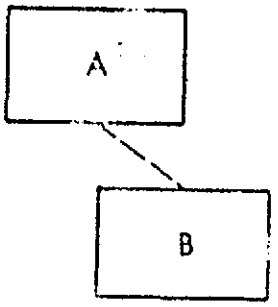
Same as above.



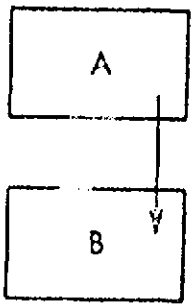
The operating environment (e.g., operating system).



Module A transfers control (without return) to the operating environment (e.g., ABEND SVC in OS).



Parallel activation.
A activates B as a parallel task (e.g., ATTACH in OS).



Module B references an externally declared symbol in module A.

In addition, most combinations of these symbols are valid.

Most of this notation is due to Constantine [4].

BIBLIOGRAPHY

1. Belady, L. A. and Lehman, M. M. "Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth," RC 3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1971.

2. Hamming, R. W. "One Man's View of Computer Science," Journal of the ACM, Volume 16, No. 1, 3-12 (January 1969).

3. "Chief Programmer Teams: Principles and Procedures," FSC 71-5108, IBM Federal Systems Division, Gaithersburg, Maryland, 1971.

4. Constantine, L. L. Fundamentals of Program System Design. Not yet published.

5. ben-Aaron, M. "Programming Systems Standardisation: A Rationale," Proceedings of the Fourth Australian Computer Conference, 307-312 (August 1969).

6. Weinberg, G. M. PL/I Programming: A Manual of Style. New York: McGraw Hill, 1970.

7. Rhodes, J. J. "Modular Planning and Control", The Computer Bulletin, Volume 16, Number 9, 320-325 (September 1970).

Myers, G. J. "Estimating the Costs of a Programming System Development Project," TR 00.2316, IBM Systems Development Division, Poughkeepsie, New York, 1972.

9. Hatfield, D. J. and Gerald, J. "Program Restructuring for Virtual Memory," IBM Systems Journal, Volume 10, No. 3, 168-192 (1971).

APPENDIX C

CHIEF PROGRAMMER TEAM
MANAGEMENT OF PRODUCTION PROGRAMMING

Reprinted from IBM Systems Journal
Volume Eleven | Number One | 1972

Chief programmer team management of production programming

by F. T. Baker

Seeking to demonstrate increased programmer productivity, a functional organization of specialists led by a chief programmer has combined and applied known techniques into a unified methodology.

Combined are a program production library, general-to-detail implementation, and structured programming. The overall methodology has been applied to an information storage and retrieval system.

Experimental results suggest significantly increased productivity and decreased system integration difficulties.

Chief programmer team management of production programming

by F. T. Baker

Production programming projects today are often staffed by relatively junior programmers with at most a few years of experience. This condition is primarily the result of the rapid development of the computer and the burgeoning of its applications. Although understandable, such staffing has at least two negative effects on the costs of projects. First, the low average level of experience and knowledge frequently results in less-than-optimum efficiency in programming design, coding, and testing. Concurrently, the more experienced programmers, who have both the insight and knowledge needed to improve this situation, are frequently in second- or third-level management positions where they cannot effectively or economically do the required detailed work of programming.

Another kind of ineffectiveness appears on many projects, which derives from the typical project structure wherein each programmer has complete responsibility for all aspects of one or a small set of modules. This means that, in addition to normal programming activities such as design, coding, and unit testing, the programmer maintains his own decks and listings, punches his own corrections, sets up his own runs, and writes reports on the status of all aspects of his subsystem. Furthermore, since there are few if any guidelines (let alone standards) for doing any of these essentially clerical tasks, the results are highly individual-

ized. This frequently leads to serious problems in subsystem integration, system testing, documentation, and inevitably to a lack of concentration and a general loss of effectiveness throughout the project. Because such clerical work is added to that of programming, more programmers are required for a given size system than would be necessary if the programming and clerical work were separated. There are also many more opportunities for misunderstanding when there is a larger number of interpersonal interfaces. This approach to multiprogrammer projects appears to have evolved naturally, beginning in the days when one-programmer projects were the rule rather than the exception. With the intervening advances in methods and technology, this is not a necessary, desirable, or efficient way to do programming today.

chief programmer teams

H. D. Mills has studied the present large, undifferentiated, and relatively inexperienced team approach to programming projects and suggests that it could be supplemented—perhaps eventually replaced—by a smaller, functionally specialized, and skilled team.[1] The proposed organization is compared with a surgical team in which chief programmers are analogous to chief surgeons, and the chief programmer is supported by a team of specialists (as in a surgical team) whose members assist the chief, rather than write parts of the program independently.

A chief programmer is a senior level programmer who is responsible for the detailed development of a programming system. The chief programmer produces a critical nucleus of the programming system in full, and he specifies and integrates all other programming for the system as well. If the system is sufficiently monolithic in function or small enough, he may produce it entirely.

Permanent members of a team consist of the chief programmer, his backup programmer, and a programming librarian. The backup programmer is also a senior-level programmer. The librarian may be either a programmer technician or a secretary with additional technical training. Depending on the size and character of the system under development, other programmers, analysts, and technicians may be required.

The chief programmer, backup programmer, and librarian produce the central processing capabilities of the system. This programming nucleus includes job control, linkage editing, and some fraction of source-language programming for the system—including the executive and, usually, the data management subsystems.

Specific functional capabilities of the system may be provided by other programmers and integrated into the system by the chief programmer. Functional capabilities might involve very

complex mathematical or logical considerations and require a variety of programmers and other specialists to produce them.

Thus the team organization directly attacks the problems previously described. By organizing the team around a skilled and experienced programmer who performs critical parts of the programming work, better performance can be expected. Also, because of the separation of the clerical and the programming activities, fewer programmers are needed, and the number of interfaces is reduced. The results are more efficient implementation and a more reliable product.

a team experiment

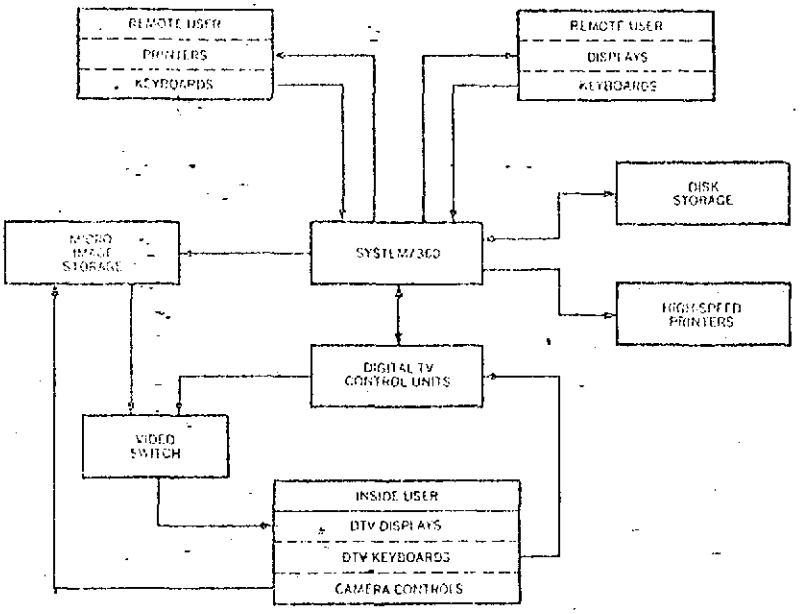
Programming for *The New York Times* information bank was selected as a project suitable for testing the validity of the chief programmer team principles. Since the programming had to interface with non-IBM programs and non-IBM hardware, this experiment involved most of the types of problems generally encountered in large system development. Besides serving as a proving ground for chief programmer team operational techniques, the project sheds light on three key questions bearing on the utility of the approach: (1) Is the team a feasible organization for production programming?, (2) What are the implications of the wide deployment of teams?, and (3) How can a realistic evolution be made? The main theme of this paper is a discussion of these questions. Before beginning, however, we present a technical description of the project, which was performed under a contract between The New York Times Company and the IBM Federal Systems Division.

Information bank system

The heart of the information bank system is a conversational subsystem that uses a data base consisting of indexing data, abstracts, and full articles from *The New York Times* and other periodicals. Although a primary object of the system is to bring the clipping file (morgue) to the editorial staff through terminals, the system may also be made available to remote users. This is a dedicated, time-sharing system that provides document retrieval services to 64 local terminals (IBM 4279/4506 digital TV display subsystems) and up to one hundred twenty remote lines with display or typewriter terminals.

Figure 1 is a diagram of the data flow in the conversational subsystem, which occupies a 200 to 240K byte partition of a System/360 (depending on the remote line configuration) under the System/360 Disk Operating System (DOS/360). Most of the indexing data and all of the system control data are stored on an IBM 2314 disk storage facility. Abstracts of all articles are stored on an IBM 2321. The full text of all articles is photographed and

Figure 1 Conversational subsystem data flow



placed on microfiche, and is accessible to the system through four TV cameras contained in a microfiche retrieval device called the RISAR that was developed by Foto-Mem. A video switch allows the digital TV display consoles to receive either computer-generated character data from the control unit or article images from the RISAR. Users have manual scan and zoom controls to assist in studying articles and can alternate between abstract and article viewing through interaction with the CPU.

Users scan the data base via a thesaurus of all descriptors (index terms) that have been used in indexing the articles. This thesaurus contains complete information about each descriptor, often including scope notes and suggested cross references. Descriptors of interest may be selected and saved for later use in composing an inquiry. Experienced users, who are familiar with the thesaurus, may key in precise descriptors directly. When the descriptor specification is complete, inquirers supply any of the following known bibliographic data that further limits the range of each article in which they are interested:

- Date or date range
- Publication in which the articles appeared
- Sources other than staff reporters from which an article has been prepared
- Types of article (e.g., editorial or obituary)
- Articles with specific types of illustrations (e.g., maps and graphs)

- Section number where an article was published
- Pages (e.g., front-page articles)
- Columns
- Relative importance of the article desired (on an eight-point scale)

Users may further specify their retrieval by combining descriptors that must appear in eligible articles by relating them in AND, OR, and NOT Boolean logic expressions.

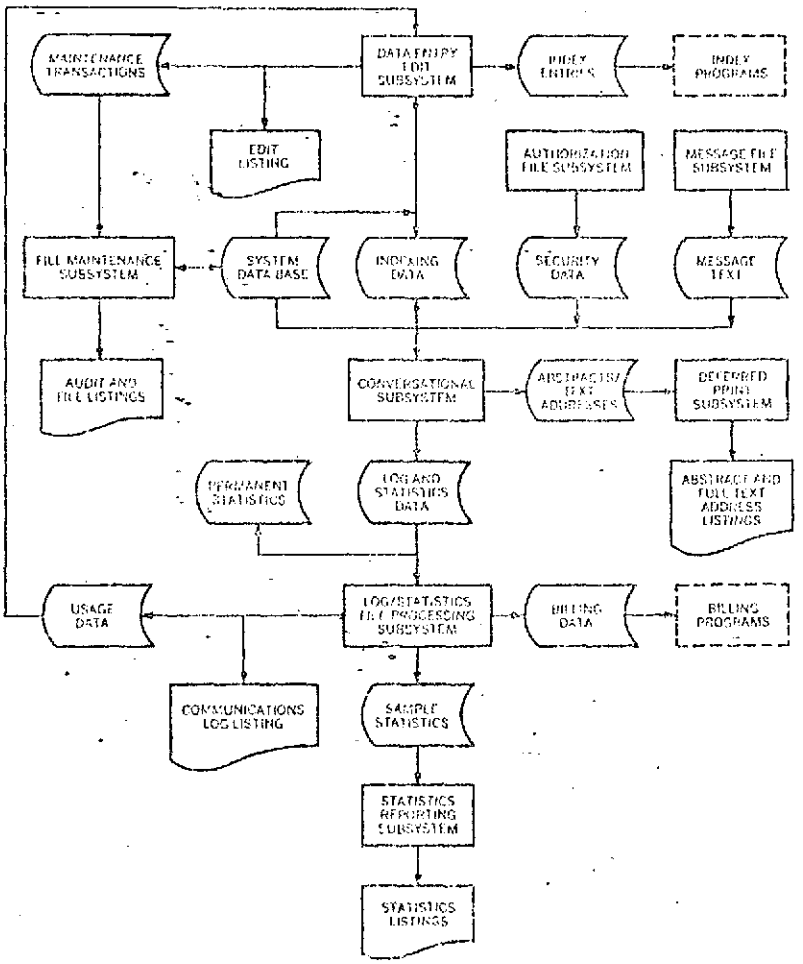
The article search is performed in two phases. An inverted index derives an initial list of articles that satisfy the Boolean inquiry statement. Articles on this list are then looked up in a file of bibliographic data and further culled on the basis of any other specified data. When the search is complete, the inquirer may elect to sort the article references into ascending or descending chronological order before he begins viewing.

Because there are only four cameras available in the RISAR, the system limits article viewing to reduce contention. Thus the inquirer views abstracts of the retrieved articles and selects the most relevant ones for full viewing when a camera becomes available. Inquirers may also request hard copies of specified abstracts and articles. Remote users cannot view the full articles directly. The references in displayed abstracts, however, identify the corresponding articles for off-line retrieval from other sources or through the mail.

A few other significant features of the conversational subsystem may be of interest. It incorporates several authorization features that inhibit unauthorized access to the system and fulfill the conditions of copyright law and other legal agreements. Inquirers who need assistance may key a special code and be placed in keyboard communication with an expert on system files and operations. This expert may also broadcast messages of general interest to all users. Several priority categories exist to allocate resources to inquirers and to control response time. In addition to inquirer facilities, the conversational subsystem allows indexers using the digital TV terminals to compose and edit indexing data for articles being entered into the system data base.

Figure 2 shows the relationship of the conversational subsystem to the supporting subsystems. The indexing data previously mentioned is processed by the data entry edit subsystem and produces transactions for entering data into or modifying the system files. Also produced is a separate set of transactions for preparing a published index. The file maintenance subsystem modifies the six interrelated files that constitute the system data base, and also prepares file backups. Security data used by the conversational subsystem to identify users and determine their

Figure 2 Information bank system



authority is prepared by the authorization file subsystem. The conversational subsystem interacts with users by presenting messages on one of three levels ranging from concise to tutorial, and the message file subsystem prepares and maintains the message file. During operation of the conversational subsystem, users may request hard copy of abstracts and/or articles. The abstracts and the microfiche addresses of the designated articles are printed by the deferred print subsystem. The conversational subsystem also transmits a variety of data on its operation to the log/statistics file, and the corresponding subsystem. A log containing a summary of operations is printed. Billing data for subscribers are passed to billing programs written by *The Times*. Usage data are passed back to be added to the data base. Usage statistics are passed to the statistics reporting subsystem, which produces detailed reports on overall system usage, descriptor (index term) usage, abstract usage, and full article usage.

Team organization and methodology

The methods discussed in this paper have been individually tried in other projects. What we have done is to integrate, consistently apply, and evaluate the following four programming management techniques that constitute the methodology of chief programmer teams:

- Functional organization
- Program production library
- Top-down programming
- Structured programming

functional organization

Since our contracts have more legal, financial, administrative, and reporting requirements associated with them than internal projects of corresponding size, a project manager coordinates these activities in all except the smallest contracts. Administrative and technical problems are jointly handled by the chief programmer and the project manager, thereby permitting the team and especially the chief programmer to concentrate on the technical aspects of the project.

A functional organization also segregates the creative from the clerical work of programming. Because the clerical work is similar in all programming projects, standard procedures can be easily created so that a secretary performs the duties of program maintenance and computer scheduling.

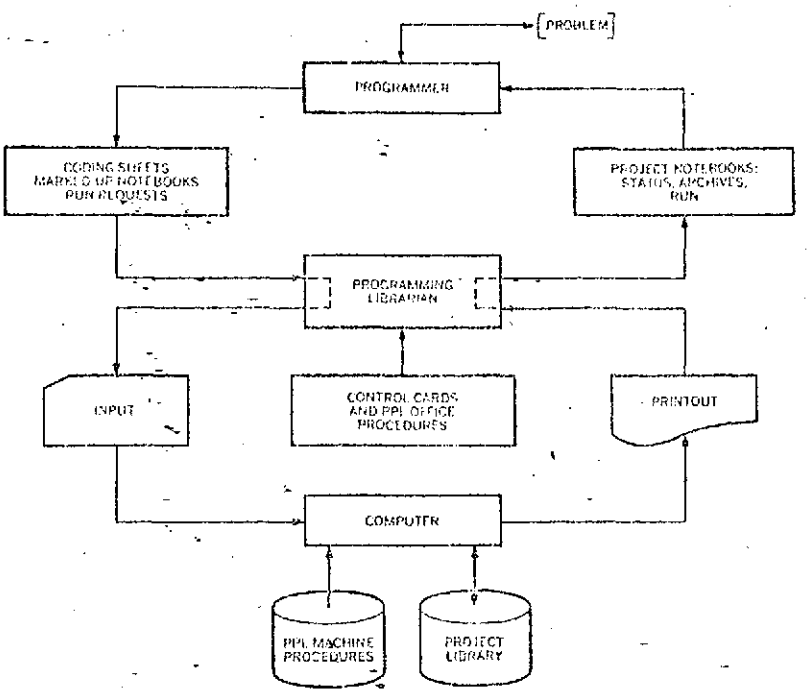
program production library

We have developed a program library system to isolate clerical work from programming and thereby enhance programmer productivity. The system currently in use is the Programming Production Library (PPL). The PPL, shown in Figure 3, includes both machine and office procedures for defining the clerical duties of a programming project. The PPL procedures promote efficiency and visibility during the program development stages.

The PPL comprises four parts. The machine-readable *internal library* is a group of sublibraries, each of which is a data set containing all current project programming data. These data may be source code, relocatable modules, linkage-editing statements, object modules, job control statements, or test information. The status of the internal library is reflected in the human-readable *external library* binders that contain current listings of all library members and archives consisting of recently superseded listings. The *machine procedures* consist of standard computer steps for such procedures as the following:

- Updating libraries
- Retrieving modules for compilations and storing results
- Linkage editing of jobs and test runs

Figure 3 Programming production library



- Backing up and restoring libraries
- Producing library status listings

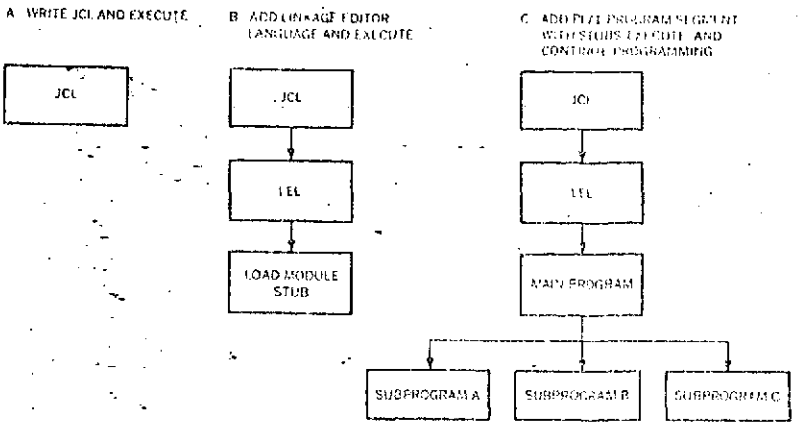
Office procedures are clerical rules used by librarians to perform the following duties:

- Accepting directions marked in the external library
- Using machine procedures
- Filing updated status listings in the external library
- Filing and replacing pages in the archives

A programmer using the PPL works only with the external library. Using standard conventions, he enters directly into the external library binders the changes to be made or work to be done. He then gives these changes to the librarian. Later he receives the updated external library binders, which reflect the new status of the internal library. The external library is always current and is organized to facilitate use by programmers. A chronological history of recent runs contained in the archive binders is retained to assist in disaster recovery. The programmers are thus freed from handling decks, filing listings, keypunching, and spending unnecessary time in the machine area.

The PPL procedures are similar to other library maintenance systems and consist solely of Job Control Language (JCL) state-

Figure 4 Top-down system development



ments and standard utility control statements. By combining standard machine procedures, standard office procedures, and project libraries, the trained librarians provide a versatile programming service that allows a team to make more effective use of its time. The PPL also assists in improving productivity and quality by providing visibility of the work, thereby allowing team members to be aware of the status of modules that they are integrating. Such visibility also permits members to be certain of interface requirements. The internal working languages of a team are the code and statements in the libraries, rather than a separate set of documents that lag behind actual status. Programmers read each other's code in order to communicate definitions, interfaces, and details of operation. Only when a question arises that cannot be resolved by reading code, is it necessary to consult another programmer directly.

top-down programming

The third technique implemented and tested is that of top-down programming. Although most programming system design is done from the top down, most implementations are done from the bottom up. That is, units are typically written and integrated into subsystems that are in turn integrated at higher and higher levels into the final system. The top-down approach inverts the order of the development process. Figure 4 depicts the essence of the top-down approach. Following system design, all JCL and link-edit statements are written together with a base system. The second-level modules are then written while the base system is being checked out with dummy second-level modules and dummy files where necessary. Third-level modules are then written while the second-level modules are being integrated with the base system. This development cycle is repeated for as many levels as necessary. Even within a module, the top-down approach is used by writing and running a nucleus of control code

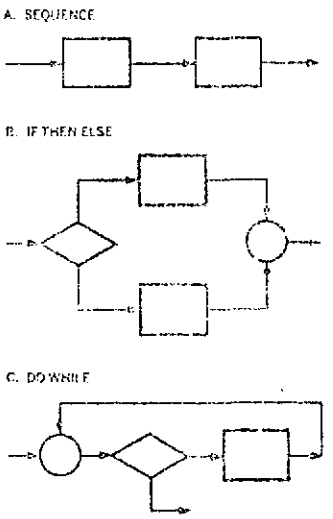
first. Then functional code is added to the control code in an incremental fashion.

structured programming

Structured programming, also used in the information bank project, is a method of programming according to a set of rules that enhance a program's readability and maintainability. The rules are a consequence of a structure theorem in computer science described by Böhm and Jacopini.[2] The rules state that any proper program—a program with one entry and one exit—can be written using only the following programming progressions that are also illustrated in Figure 5.

A. Sequence
B. IF THEN ELSE
C. DO WHILE

Figure 5 Structured programming



Although these rules may seem restrictive and may require a programmer to exercise more thought when first using them, several advantages ensue. With the elimination of GO TOs, one can read a program from top to bottom with no jumps and one can see at a glance the conditions required for modifying a block of code. For the same reason, tests are easier to specify. Further, the rules assist in allowing a program unit to be written using the top-down approach by writing control statements first and then function statements. The use of CALLs to dummy subroutines or INCLUDEs of empty members permits compilation and debugging at a much earlier stage of programming. Finally, if meaningful identifiers are used, a program becomes self-documenting and the need for lengthy comments and flow charts is reduced.

Conventions to support the use of structured programming are required. A set of rules has been developed to format source code so that indentation corresponds to logical depth. If extensive change is necessary, a program is available to reformat the source code.[3] To make minor changes such as moving some code a few columns, a utility program may be written or an existing one modified. Also, the lengths of individual blocks of source code are small to enhance readability and encourage a top-down approach. The objective is to have no block exceed a single listed page, or about fifty lines. Finally, by extending the range of structured programming progressions, efficiency of object code can be significantly improved, and source code readability is not impaired. Thus, iterative DOs with or without a WHILE clause and a simulated ALGOL-like CASE statement based on a subscripted GO TO statement and a LABEL array were permitted in our project.

Structured programming has been described in terms of languages with block structures such as PL/I, ALGOL, or JOVIAL. It is possible to introduce a simulated block structure into other

types of languages and then to develop structuring rules for them also. This has been done for System/360 Assembler Language, a low level language, through a set of macros that introduce and delimit blocks and provide DO WHILE, IF THEN ELSE and CASE-type figures. Further, if the long identifiers permitted by Assembler H are used, the source code is even more readable.

System development

This section discusses how the previously described techniques have been used in developing the information bank. The project was originally staffed with a chief programmer, a backup programmer, a system analyst (who was also a programmer), and a project manager. Since a project requirement was that the information bank operate under the System/360 Disk Operating System (DOS/360), the backup programmer began developing a version of the programming production library (PPL) that would operate under DOS/360. In parallel, the chief programmer and the system analyst began developing a detailed set of functional specifications. The first product of the team was a book of specifications that served as a detailed statement of the project objectives.

The team, at this point, reoriented itself from an analysis group into a development group, and a programmer technician was added to serve as a librarian. The system analyst began detailed design of system externals, such as the messages, communication log, and statistics reports. The chief programmer and backup programmer worked together on designing the various subsystems and their interfaces.

file maintenance subsystem

Since the system is heavily file oriented, efficient retrieval and the capability of adding large volumes of new material daily were requirements. Therefore, the chief and backup programmers initially emphasized the development of an interrelated set of six files that provide the necessary file attributes. Declarations of structures for these files were the first members placed in the library. Detailed file maintenance and retrieval algorithms were developed before any further design was done.

A substantial amount of data already existed on magnetic tape. Therefore, to begin building files for debugging and testing the system, it was desirable that the file maintenance subsystem be developed. This subsystem was designed to consist of two major programs and several minor ones. The chief programmer and backup programmer each began work on one of the major programs. Working in top-down fashion, control nuclei for each major program were developed. Functional code was gradually added to these nuclei to handle different types of file mainte-

nance transactions until the programs were complete. The minor programs were then produced similarly.

Because of the early need for the file maintenance programs, an independent acceptance test was held for this subsystem. One of the functions performed by the backup programmer was the development of a test plan that specified all functions of the subsystem requiring testing and an orderly sequence for performing the test using actual data and transactions. An indication of the quality achievable by the chief programmer team is afforded by the fact that no errors were detected during the subsystem test. In fact, no errors have been detected during fifteen months of operation subsequent to the test.

data entry subsystem

While the file maintenance subsystem was being developed, the chief programmer and system analyst designed an on-line system for keying and correcting indexing data destined for information bank files and for *The New York Times Index*. This indexing system became the data entry subsystem and additions to the conversational subsystem. The *Index* had previously been prepared by a programming system from data obtained by keying a complex free-form indexing language onto paper tape. The existing language was, therefore, extended to include the fields needed by the conversational subsystem and formalized by expressing it in Backus-Naur form. Because it was likely that the language would be modified as the project evolved, we decided to perform the editing of indexing data using syntax-direct techniques. (Another programmer was added to the team to develop the data entry subsystem around the syntax-directed editor.)

After the file maintenance subsystem had been delivered and the externals of the system specified, the system analyst programmed the authorization file subsystem, the message file subsystem, the log/statistics file processing subsystem, and the deferred print subsystem. (Another programmer was added, who wrote the statistics reporting subsystem.)

The chief programmer and backup programmer developed the conversational subsystem. Again, operating in top-down fashion, first programmed was the nucleus consisting of a time-sharing supervisor and the part of the terminal-handling package required to support the digital TV terminals. This nucleus was debugged with a simple function module that echoed back to a display material that was typed on the keyboard. After the nucleus was operational, development of the functions of the retrieval system itself commenced. System functions were programmed in retrieval order, so that new functions could be debugged and tested using existing operational functions, and an inquiry could proceed as far as programming existed to support it. All debugging was done in the framework of the conversa-

tional subsystem itself, and because of the time-sharing aspects of the system, several programmers could debug their programs simultaneously. The ability to modify tests as results were displayed at a terminal was helpful in checking out new code. Two programmers were added to the team to write functional code. A third programmer was added to extend the terminal-handling package for the 2260 and 2265 display terminals, and for the 2740 communication terminal. These programmers rapidly acquired sufficient knowledge of the interface with the time-sharing supervisor to write functional code despite their short participation on the team.

system testing

During this development process, the backup programmer prepared a test plan for the rest of the system to be used with realistic inquiries for the test. Although some errors were found during a five-week period of functional and performance testing, all were relatively small, and did not involve the basic logic of the system. Most errors were found in the functional code that had been most recently added to the system and had been the least exercised. The performance parts of the testing measured both sustained load handling and peak load handling. In spite of the fact that the performance tests were run on a System/360 Model 40 with three 2314 disk storage facilities as files, instead of on the System/360 Model 50 with seven disk storage facilities for which the performance objectives had been developed, performance objectives were successfully met.

Productivity

A key objective of the chief programmer team approach was to demonstrate increased productivity of the team over an equal number of conventionally organized programmers. This section discusses data on the productivity of the team and their strategy for using their time. Typical productivity measures are computed to facilitate comparison with other projects. Table 1 breaks down the staff months applied on the project, and Table 2 displays measures of amounts of source code produced.

Standardized definitions have been used in preparing these tables and achieving comparable measures of productivity. *Source lines* are eighty-character records in the library that have been incorporated into the information bank and consist of the following kinds of statements:

- Programming language
- Linkage-editor control
- Job control

Source coding has been broken into the following three levels of difficulty, which are summarized in Table 2:

Table 1 Analysis of project staffing by time and type of work

Work type	Staff time (man months)											
	Chief	Backup	Analyst	Programmer 1	2	3	4	5	Technician	Manager	Sec'y	Total
Requirements Analysis	2.5	1.0	8.0	0.5	–	–	–	–	–	–	–	12.0
System design	4.0	4.0	4.5	1.0	–	–	–	–	–	–	–	13.5
Unit design, programming, debugging, and testing	12.0	14.0	10.0	13.0	4.5	2.8	3.7	4.5	–	–	–	64.5
Documentation	2.0	2.0	4.5	1.5	0.2	0.2	0.3	0.3	–	–	–	11.0
Secretarial	–	–	–	–	–	–	–	–	–	–	7.0	7.0
Librarian	–	–	–	–	–	–	–	–	5.5	–	2.0	7.5
Manager	3.5	2.0	–	–	–	–	–	–	–	11.0	–	16.5
Total	24.0	23.0	27.0	16.0	4.7	3.0	4.0	4.8	5.5	11.0	9.0	132.0

Table 2 Lines of source coding by difficulty and level

Difficulty	Level High	Low	Total
Hard	5034	–	5034
Standard	44247	4513	48760
Easy	27897	1633	29530
Total	77178	6146	83324

- *Easy coding* has few interactions with other system elements. (Most of the support programs are in this category.)
- *Standard coding* has some interactions with other system elements. (Examples are the functional parts of the conversational subsystem and the data entry edit subsystem.)
- *Difficult coding* has many interactions with other system elements. (This category is limited to the control elements of the conversational subsystem.)

Source coding types have been categorized as one of the following:

- *High-level* coding in a language such as PL/I, COBOL, or JCL
- *Low-level* coding such as assembler language and linkage-editor control statements

Table 3 presents some simple measures of programmer productivity based on the same coding used for producing Tables 1 and 2. The first row includes work done on unit design, coding, debugging, and acceptance testing. The second row summarizes