A SURVEY OF PROGRAMMING METHODOLOGIES

by

JULIA FU LEE, B.A.

May, 1975 TR-47

This paper constituted in part the author's thesis for the M.A. degree at The University of Texas at Austin, May 1975.

Technical Report 47

THE UNIVERSITY OF TEXAS AT AUSTIN

DEPARTMENT OF COMPUTER SCIENCES

ABSTRACT

This paper examines how the process of software development may be improved through the use of the following three program methodologies.

Modular Programming is a technique which breaks down the entire workload of a project into small units of work and makes the project more manageable.

The Chief Programmer Team is an organizational concept centered around a team leader who designs the system, writes the critical code, and supervises the project with the support of specialists.

Top Down Structured Programming is a disciplined approach to developing software. It applies the concepts of top down design and structured programming to produce better quality programs.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

CHAPTER 1

INTRODUCTION

1.1 THE PROBLEM OF SOFTWARE COST

Over the last decade, the major trend observed in data processing centers has been the steady increase in software costs while hardware costs have decreased dramatically.

Due to technological advances, hardware has been capable of providing higher speed and lower cost per transaction on a continuing basis. This has come about through tangible improvements in all parts of the system hardware. CPU speed is up through the use of faster circuit technologies and denser packaging. Semiconductor memories with their higher densities offer faster access and more storage per unit of space at lower cost than the older ferrite core memories. Peripheral devices have higher data transfer rates, are packing more data per unit of space on the recording media and as a consequence, provide extended mass storage capabilities at lower and lower cost per bit. All these factors have resulted in higher performance and more reliability in systems with significantly larger memory and

greater mass storage capacity at costs substantially lower than that of equivalent equipment from the previous generation.

On the other hand, although software technology has taken giant strides in the last ten years, it has been unable to keep pace with the advances in hardware. The problem seems to be that software cannot take advantage of the reduced computing costs of new hardware without boosting its own costs. What accounts for this phenomenon? First, the new computing systems are complex and sophisticated and provide highly sophisticated capabilities. Programming them effectively requires a thorough understanding of these capabilities. Often, this level of understanding must include an awareness of why these new capabilities are being provided and also how they overcome the inadequacies of previous systems. Moreover, as new systems are introduced, each provides new and more powerful options that the programmer must discern and master. This is especially true if he or she is working at the assembly language level. It takes considerable knowledge and experience, for instance, to make an intelligent decision regarding which of the many available data access methods will be the most efficient in handling a particular I/O problem. While it is true that the new systems have removed the burden of coding such onerous and tedious tasks

as data access methods, or with the advent of virtual memory systems, the construction of overlay schemes, they have spawned a profusion of confusing options that the programmer must learn and comprehend. These options have freed the programmer from concentrating so much of his effort on dealing with the mechanics and limitations of the system, and when used properly, give him the opportunity to make intelligent and creative choices regarding the solutions to his systems or applications programming problems. They reduce the tedium associated with programming and open up new vistas for the creation of more complex and sophisticated programs. Inherent in this reasoning, however, is the assumption or rather the expectation that the programmer possesses sufficient depth in his understanding of system capabilities to come up with an efficient and refined piece of software to do the job at hand. Thus, the basis for effective use of the new systems assumes reliance on the higher level of skills and knowledge which are the mark of the good experienced programmer.

Second, the increased capabilities of the new machines mean that it becomes both feasible and cost effective to greatly expand the range of applications that can be performed by these systems. This means that, as

time progresses, more and more complex applications will be tackled, and this in turn will result in more and varied forms fo software to write, to test, and to maintain. The trend in this direction is already very apparent from the published statistics. Ten years ago, software accounted for 40% of the total cost of a data processing operation. Today, it has reached 70% and by the early to mid-1980's, it could be close to 90% as hardware costs continue to go down and software and people costs continue to go up. In addition, two other factors are contributing strongly to the software cost spiral. One is the expansion in the use of large systems. Software is a critical part of these systems, and its high costs are incurred not only during development, but often throughout the entire life cycle of the machine. Some installations have reported that 40% or more of their software effort has gone into maintenance. The other factor that compounds the cost of the software life cycle is the incompatiblility between computers. Migration expenses are a major consideration when a change in hardware is planned or contemplated, and the performance loss suffered while running in a degraded mode during emulation of an old machine on a new system adds another software cost that warrants attention. This means that when migration costs are being estimated, both the one time start up costs of switching over from one

system to another as well as the continuing costs of emulation should be included.

What can be done about the incessant rise in software costs? To find an answer, let us take a closer look at the factors that influence these costs.

1.1.1 THE FACTORS THAT INFLUENCE SOFTWARE COST

A. Personnel

People costs are rising every year. The strong demand for programmers in the past has greatly boosted their salaries and strained data processing budgets. The goal of any data processing center should be the effective utilization of their programming personnel. The key measure of the effective use of programming talent has been programmer productivity, and the key issue has been how programmer productivity can be improved. Numerous studies have concentrated on ways to increase the productivity of the people involved in the process of creating or maintaining software. The main areas that these studies single out as needing improvement are programming languages, documentation standards, programming methodology, and project management.

B. Management

Management is one of the biggest factors in the success or failure of software projects. It is management's role to make trade-offs, to evaluate new methodologies and tools, and to make vital decisions on schedule changes and modifications to specifications during development. The approaches taken and the decisions made on these items will directly affect the cost and outcome of the project.

The most common problems facing management today deal with controlling the quality of programs. Most programs produced have not been tested throughout the full range of possible inputs and, because of this, contain more errors than they should. How has management allowed this to happen? There are several contributing factors. First some data processing managers have had no or only a smattering of experience in the data processing field. Then there are the previously experienced managers who have failed to keep themselves up-to-date. Managers in these categories could hardly be expected to have the level of understanding needed to maintain a tight grip on program development and testing procedures. A second factor is the intuitive feeling many managers have that not using quality control procedures appears to be less expensive than using them. Those holding this viewpoint, however, ignore the

secondary costs of time lost correcting inadequately tested programs as well as the great expense of attempting to maintain an interrelated collection of hastily written and poorly documented programs. A third factor can be attributed to the widespread practice of the dispersal of responsibility in data processing projects. Often, when this happens, there is no way of assigning final responsibility for the project, and the failure to meet project goals results in an exercise of "buck passing". A fourth element is that at present, there are no good ways for estimating the costs of producing software. Frequently, it is impossible to ascertain how much a piece of software costs after it has been produced. With no reliable way to estimate how much a piece of software will cost, it is just not possible to accurately determine how different management policies or techniques might have altered the cost.

This is a sad commentary on the status of dada processing management. There is no denying that serious problems exist. Nevertheless, there are ways to improve the situation. As far as management is concerned, the type of structure taken on by the programming organization is very important. It has a definite effect on the final product quality and the productivity of the programming

team. This will be one of the main topics covered in this paper.

C. Hardware

Software development has been hampered by the meager level of communication between the hardware and software communities. The lack of a really open dialog has prevented both sides from understanding what the other's major problems are and from working out design strategies that would be of mutual benefit. Evidence of such limited communication is the virtual absence of computer architecture research in the universities and the existence in the marketplace of hardware with such poorly designed primitive functions that they require the repetitious application of costly and error-inducing programs to provide the basic computational functions.

Another major problem area fostered by poor communications and the lack of hardware - software interface guidelines or standards is the incompatibility between computers. Programs written for one computer must often be rewritten in order to work on another machine. This factor must be given major consideration when making the decision to migrate from one machine to another, for once migration is underway, the only alternatives are to accept

the recourse of costly reprogramming or to forego the ability to take full advantage of the reduced computing costs of the new hardware. Neither of these choices is in any way appealing or attractive to the user.

All of these hardware induced problems are deficiencies that we have tolerated up to the present time for the lack of anything better. It seems that in today's world we should have progressed sufficiently in the evolution of hardware and its supporting microprograms to be able to formulate a more central architectural statement of system capability. Properly modularized designs of logic circuitry and microprogramming could produce a good and flexible set of primitives. These in turn could permit the design of system architectures more tailored to specific user applications from the same basic hardware system. This is certainly one of the ways hardware architecture can help alleviate the growing software complexity problem.

To bring this goal within reach, work has to be done to identify and define functionally complete sets of primitives for such application areas as:

* Data bases
* Communications applications
* Direct support of major programming languages (COBOL, PL/1, FORTRAN, APL, etc.)

These results should be backed up by the implementation in hardware/firmware of generalized data structures such as stacks, linked lists, and rings. The benefits to be derived from all these activities will be more general and possibly greatly simplified programming for the operations in the major areas of computer applications.

1.1.2 SOFTWARE QUALITY

So far the discussion has centered on the areas that affect software cost. One other characteristic of software goes hand in hand with any discussion of cost, and that is software quality. What makes up this quality and what does its absence or presence mean to the user? What are the factors that influence program quality? The following definitions should help to answer these questions.

A. Clarity

Clarity in the program code and its documentation is measured by the effort that a person, other than the author, has to make in order to understand it. The clearer the code and documentation, the smaller the effort required for comprehension.

B. Cost

The total cost involved in using the program during the life cycle of the product. This includes the

initial cost of producing the program plus the follow on costs of maintaining or modifying it to meet new or changing needs.

C. Maintainability

Maintainability is measured by the level of effort needed to support the program after it has gone into production mode. Hopefully, the program has been written with sufficient clarity and modularity and been tested thoroughly enough to require a low level of effort to keep it running well.

D. Modifiability

Modifiability is the characteristic that describes the effort required to alter a program in order to adapt it to the user's changing needs. A well thought out and carefully written program will usually leave open a number of options for change and have the flexibility to allow the implementation of most changes without requiring costly modifications throughout the whole program.

E. Performance

Performance refers to any factor in the program that influences the consumption of resources. Ideally, the program should be designed to minimize the consumption of resources and still perform at high speed. Normally, some compromise is reached between these two conflicting goals due to either a physical limitation in the system or the

unacceptability of running very long jobs. Performance generally suffers when there is a lack of optimization between the allocation and use of computer resources and the goal of a short run time.

F. Portability

Portability is measured by the amount of effort expended in moving a program over to a different installation, a different machine, or a machine with a different operating system. Portability is achieved by making the program as machine independent as possible, and this usually means coding the program in a widely used high level language. The advantage in doing this must be balanced against a loss in computing efficiency.

G. Robustness

Robustness describes the ability of a program to continue to perform its functions in spite of some violation of the assumptions in its specifications. A program demonstrates this quality if, for instance, it can ignore a limited number of bad data records interspersed with good data records and upon encountering the bad data, can recover and continue with its processing of the good data.

1.2 OBJECTIVES

The increased capabilities of the new generation

of data processing systems have brought a wide range of new applications within their reach. As more and more complex applications fall within the realm of these systems, the programs required to implement these applications have also become correspondingly complex. This phenomenon has stimulated intense interest in examining the processes involved in program development and the techniques used in project management.

The remainder of this report will focus on these two areas. It will deal with the techniques and disciplines that can be brought to bear against the problems of high software cost and poor software quality. The objective will be to show how

* design philosophies
* the breakdown of the programming task
* the structuring of programs
* the organization of the programming team

can affect and, if properly integrated, can reduce software costs and improve both productivity and software quality.

Chapter 2 discusses the concept of modular programming. It explains how a program design can be modularized into small functional units and examines the potential advantages to be gained from using this programming methodology.

It describes the possible gains

(1) to management in the areas of estimation and project control,

(2) to program development in the areas of design, testing, and integration, and

(3) to program quality in the areas of program reliability and maintainability.

It also covers the factors that determine whether modular programming is suitable for a particular installation. Topics discussed are the installation's size, its normal kind of workload, and the skill levels of its personnel.

Chapter 3 describes the programming organization known as the Chief Programmer Team. This organization is led by and operates under the close supervision of the chief programmer and his assistant, the backup programmer. Both of these individuals are highly qualified programming specialists. They are responsible for the design of the system and for reviewing the implementation work done by the other team members. Often, they will code the critical routines themselves.

This organization emphasizes strong technical leadership over a small and experienced group of professionals. The structure has well defined positions and individuals are treated as specialists. Chapter 3

describes how the Chief Programmer Team is organized, its mode of operation, and the benefits to be derived from this organization in terms of program productivity, reliability, and maintainability. Installation related factors are also discussed.

Chapter 4 discusses the concept of top down structured programming. This concept combines the discipline of top down design with the discipline of structured programming. The result is a well integrated, well organized system design based on a tree-like structure (the influence of top down design) and an implementation that is in terms of small, easy to follow routines coded from a restricted set of program figures and options (the influence of structured programming). The benefits of this disciplined approach are discussed from the management standpoint of project control and from the technical standpoints of program design, testing and integration, reliability, and maintainability. In addition, a fairly complete description of top down design and structured programming is provided.

Chapter 5 summarizes the material from Chapters 2, 3, and 4. It compares the concepts of modular programming, the Chief Programmer Team, and top down structured programming in terms of their advantages and disadvantages.

It concludes with the factors that programming installation should consider before making a decision on adopting any particular programming methodology.

1.3 REFERENCES

1. Boehm, B. Software and Its Impact: A Quantitative Assessment. Datamation. May 1973, Vol. 19, No. 5.

Boehm assesses the effect of software problems on an Information Processing Center at a U.S. Air Force command and control site. He charts the trends in hardware expenditure for the last decade. He gives a very complete and quantitative appraisal of the software costs encountered during the same period.

2. Dijkstra, E. The Humble Programmer. Comm. ACM. October 1972, Vol. 15, No. 10.

This is an extract of the 1972 ACM Turing Award presentation to Dr. E. W. Dijkstra at the ACM Annual Conference in Boston. In this paper Dijkstra discusses the evolution of software and the impending crisis in software because progress there has not kept pace with improvements in hardware. Among other things, he covers the birth and impact of high level languages and emphasizes that quality, reliability, and cost will be the primary software considerations of the future.

3. Goldberg, J. (ed) Proceedings of a Symposium on the High Cost of Software. Proceedings of a Symposium Sponsored by the Air Force Office of Scientific Research. Stanford Research Institute, Menlo Park, California. SRI Project 3272. 1973.

This paper makes recommendations regarding the kinds of research that are needed in order to solve the software crisis.

4. Naur, P. and B. Randell (eds) Software Engineering. Report on a Conference Sponsored by NATO Science Committee. 1968. Garmisch, Germany. Scientific Affairs Division, NATO, Brussels 39, Belgium.

This report summarizes the results from a conference concerned with software development, production, utilization, and services. The purpose of this meeting was to build a practical discipline for software based on theoretical foundations. This document is a good reference and one of the earliest to address many of the prominent problems in software.

5. Parnas, D. Some Conclusions from an Experiment in Software Engineering Techniques. Proceedings AFIPS 1972. Vol. 41.

This report describes how software experiments should be designed and evaluated. It is a good reference work.

6. Weinberg, G. The Psychology of Computer Programming. Van Nostrand Reinhold Company, New York, N.Y. 1972.

This book is easy to read, informative, and contains numerous insights into the world of computer programmers. It addresses the human dimensions of programming performance.

CHAPTER 2

MODULAR PROGRAMMING

2.1 INTRODUCTION

Within the data processing organization, one department or group that usually rates very poorly is the programming group. It has gotten its notoriety from three traits that seem to plague programming groups. One is the failure to adhere to or meet specifications, the second is the consistent failure to meet deadlines, and the third is the release for use of programs that are not thoroughly debugged. These problems occur because of the lack of a really effective way for making estimates in terms of time or resources, specifications that often turn out to be ambiguous, the lack of trained personnel, mis-management of the programming area, and poor control over and poor evaluation of the design and coding of product programs.

In any programming group, the management team has to rely on precise and timely progress reporting as it relates to the schedules and estimates for the work involved. This has not been the norm in the data process-

ing field even during the development of complex software systems, especially when the conventional or monolithic programming approach is used. In this kind of environment, the development phases of a program are strictly serial. Progress in terms of actual program goals met or functional completeness is difficult to measure until the whole program is well into its testing phase. If major design problems are revealed by the test cases at this time, management will be in for a highly unpleasant surprise since none of the feedback reaching them during development would have touched on these problems (because at that time no one else was aware of them either). Management would be unprepared to meet such a major setback so late in the game. In all probability, their reaction to this situation would be to toss out all their old schedules, put the newly discovered problems under study, and begin anew on drafting a new set of schedules. Such unexpected setbacks are a shock to management and the programming team as well.

In the conventional programming environment, the first step of any project is the design phase which is undertaken by the system analysts. It is their responsibility to interface with the user, to initially come up with the specifications of the program, and later to design

the program according to these specifications. Once the design is completed, the programming team takes over. They read the design document, interpret the specifications contained in it, and translate these into code. After all the coding is finished, there is a unit test phase. This is followed by a system test during which the programming team's product is subjected to as thorough a set of tests in an environment as close to real life as possible. It is at this stage of the development cycle that all kinds of problems, both major and minor, are discovered and have to be resolved. This is also the time when the appearance of a design problem can make a major impact on the schedule.

If, during the design phase, the entire workload of the project had been divided into small units of work, management would have been able to get a more accurate reading on the status of the program. Tracking the project's progress would have been easier and more precise because tracking would have been done on each of the work units rather than on the program as a whole. A comparison of the status of the various work units against a set of checkpoints would give a good indication as to whether the project was on or off schedule. Also, more than likely, the magnitude of the problems found during testing would correspond in size with the smaller pieces. Moreover,

there should be fewer surprises encountered at system test time because the large amount of effort spent in modularizing the program should produce a well thought out design.

This technique of dividing the workload into small work units is called modularization. Modularization is a very useful management tool. It breaks down the overall program into smaller, simpler units which are easier to track and control, and when the statuses of the individual units are combined together, they help management to secure the precise status of the entire project. In actual use, the program modules are short routines which are connected to each other via entry and exit points and communicate with one another through common data areas and passed data arguments.

2.2 DEFINITION

Modular programming is a way of developing programs as a set of interrelated individual units called modules that are constrained to follow a set of rules which control their characteristics. The program is segmented in such a way that each unit performs a clearly defined task. This permits each unit to be coded, compiled, tested, and executed as an individual entity.

The objective behind modularization is to make large systems more manageable by dividing them into smaller and more controllable pieces. This makes it easier to track the progress of the program during development and pinpoint problem areas quickly so that more resources can be channelled into these areas. Modularization also permits more efficient use of personnel in an even more basic way. Since the program is broken down into individual units that can stand on their own, all the modules can be implemented and tested in parallel. This has the effect of smoothing out the workload and allows the members of the team to schedule themselves with greater precision. Another advantage of modularization is that it prevents the duplication of effort. When this concept is applied, a single function, regardless of how many times or places it is used, needs to be written only once.

Modularization is a good way of organizing the programming of complex software systems. The success of this technique depends directly on how carefully the modules are chosen. Since the definition of each unit or module is so important, it is often assigned to a senior member of the programming team. Working from the system specifications, he will segment the system into modules such that each one will represent either a single logical

function or a number of small but related logical functions. In order to achieve high modularization, he will also try to minimize the connections and dependencies among the various modules.

2.3 CHARACTERISTICS

Modular programming is identified by a number of characteristics that are highly desirable from the viewpoint of software development. These consist of the following items:

A. Modules are independent from one another. Each module is considered to be an independent unit, and each one can be developed separately by different programmers. During the design phase, individual sets of specifications are produced for the implementation of each module.

B. Modules perform a single function or a small number of related functions. This limits the size of the module and insures that it will be a fairly independent entity. Each module can call other modules or can be the target of a call from another module.

C. Modules have standard interfaces. Such an interface provides a standard method for communicating among modules and can be well documented. The standard sets

well defined rules regarding the passage of information from one module to another and in the usage of save areas. Its purpose is to avoid the creation of modules with mismatched interfaces. Examples of standard interfaces are:

* Upon entry into a module, it will always store the contents of registers and then restore the same registers just prior to exiting.
* All modules will return control to the calling module in the same way.
* Parameters are always passed between modules in the same way.
* All modules will adhere to a single classification for entry and exit points and will abide by a common set of procedures in case of failure.

These standards are important, for without them, there would be many linkage errors. Most of the high level languages available today provide standard interfaces or provisions for these interfaces in the linkages between routines.

D. Modules generally have only one entry point and one exit point. This rule is followed even though some programming languages provide multiple entry and exit

points. The reason for this is to maintain clarity in the code and to reduce the risk of inducing an error when changes must be made.

E. Modules may be tested independently of one another. This is possible because of the semi-autonomous nature of the modules. As a consequence, it becomes feasible to test a whole program thoroughly by the separate testing of each module, and testing can begin at an earlier stage in the project.

F. Modules are small in size. They derive this attribute from the intentional effort to limit the scope of their functional capability. This has the advantage of keeping modules down to a manageable size.

2.4 THE PROBLEMS THAT MODULAR PROGRAMMING ATTEMPTS TO SOLVE

During the era of the first and second generation machines, large computer programs had to be segmented because they exceeded the memory capacity of these machines. Most large programs were run as overlays so that only certain portions of them would reside in core and be active at any one time. It was left to the ingenuity of the programmer to design his overlays in such a way that he could move them into and out of memory

without causing a substantial degradation in processing. With the introduction of third generation computers, the capacity problem disappeared as did the need for overlays (or so people thought). Programmers began to tackle new and more challenging applications while managers looked for new programming tools that could improve productivity and provide a disciplined approach to the planning and implementation phases of programming projects. This forced them to look again at the techniques of segmentation and forming overlays. These techniques appeared to possess the basic structure for applying the discipline being sought, and they were revived to become the precursor of what we know as modular programming today. From these beginnings, the ideas and techniques behind modular programming have grown and been refined to the point where they now serve a useful role in the management of projects. They can help managers in six important areas of their work. These are:

(1) Estimating and Scheduling

(2) Project Control

(3) Program Design

(4) Program Testing

(5) Program Reliability

(6) Program Maintenance

The following sections will describe how each of

these areas is affected by modular programming.

A. Estimating and Scheduling

Estimating the size of a job in terms of the resources needed to develop and support it through a period of time has been one of the most difficult tasks facing the managers of data processing centers. This is where many managers have failed time after time because they have lacked the techniques to make accurate estimates. When modular programming is used, however, the job of making estimates becomes significantly easier to handle. The whole programming effort is divided into many modules or work units. Since these modules are small in size, the module production rate for the installation or for each programmer can be calculated based on past performance. At this point, a rough estimate of the resources needed to perform the job can be computed by multiplying the total number of modules by the average time required to produce a module. Refinements to this estimate may be made by modifying the production rates associated with each module to account for variations in complexity and in programmer skill. This requires more detailed analysis, but yields more accurate figures.

Throughout the entire project, the module production or productivity rate for the installation is

updated and refined continuously based on the progressive improvement in programmer skills and, more importantly, on the actual data coming in from the completion of tasks. (If more detail is required, this procedure may also be followed for updating each programmer's individual productivity rate). Herein lies one of the most impressive aspects of modular programming - namely that each estimate is under continuous examination and refinement. The validity of each estimate can be measured as the project proceeds since the completion of each module occurs in a relatively short time. Progress for each module can be determined by comparing actual rates against estimated rates in the development cycle. In this way, attention is drawn to problem areas early enough so that if adjustments or corrections are required, they can be made without imposing a great impact on the overall project. This contrasts sharply with the non-modular programming environment where productivity is usually based on the total lines of code produced during the entire project, and, in which case, progress is not easy to measure until the whole program nears its conclusion. Disclosure of any problems at this late and advanced stage will as a rule cause a significant impact on the schedule for the whole project.

Productivity rates based on the work done at one installation are generally applicable to that individual installation only. They cannot be applied to another installation with any degree of accuracy because of a number of factors. These include differences in the skills and size of the programming teams, differences in approaches used for implementation, and differences caused by using different hardware configurations or systems - for example, batch processing versus on-line or remote processing. The best way to arrive at productivity rates for any particular installation is to derive them from figures based on general or modular programming experience gained over a period of time at that site.

B. Project Control

Project control is concerned with utilizing estimates as the basis for tracking progress, measuring performance, and controlling resources. In modular programming, the entire project is broken down into modules so that progress tracking is carried out by watching over the status of each unit. When these individual statuses are combined together, they essentially describe the condition of the whole project.

The modular programming environment provides a great deal of flexibility during the program implementation phase. Using the information collected for each module,

managers can chart performance and progress. The results found in these two areas are valuable aids in helping them to exercise their prerogatives in switching job duties around and in evaluating and updating their priorities. Managers can react quickly to any problems that may occur in any of the modules. The can focus attention and assign resources to these units before the problem gets out of hand. Usually, the problems are discovered early enough and are dealt with soon enough so that harm to the total project is minimized. Modular programming also helps managers to react to personnel problems, especially in the area of job assignment. If, for example, a programmer leaves, is ill, or is away from his job for any other reason, the manager can bring in a new person to fill the opening. The substitute can begin working almost immediately since the amount of re-learning is small even if he has to start from the beginning and rewrite the entire module. In addition, if the new person were to write only a single module, he would still be contributing to the project. It is factors like these which demonstrate that in a modular programming environment, managers can exercise positive control over project development instead of finding themselves limited to just progress recording.

C. Program Design

The technique of modular programming embraces a

very important approach to the development process: Modular Design. Modular design means that each module is designed, flowcharted, coded, tested, and documented separately.

For modular design to be effective, it is essential that the design phase be completed before programming begins. During this phase, important work is done to determine the logical structure of the total program. This includes defining the logical structures of each of the component modules as well as specifying the interrelationships among the modules. The design phase is clearly a strong asset of modular programming. It forces time to be spent during the planning stages to work out the design of a total system, to break it down into its components, and to come up with the detailed logical design for each of these components. The effort expended here ensures that major design problems will not be overlooked only to surface later on in the coding or testing stages of the project.

The design phase is usually the responsibility of the senior level people in the programming team. This is the most important part of the whole effort and requires experienced and talented people to handle the wide-ranging and varied tasks involved. First, the designers must go over the specifications carefully with the user to determine if their understanding of what is required is really

what the user is requesting. Second, they must determine how realistic the specifications are and how closely they can be met. They should point out wherever possible, modifications to the specification which will affect points that are relatively unimportant to the user, but can mean a substantial reduction in the programming effort. Third, they must determine what features form the core of the user's requirements and what are in a sense 'optional' so that development can concentrate initially on the essential elements and then shift to the non-essentials. If this is done, the program product will become available to the user at an earlier time. He will have a basic program running and will be able to pick up additional functions and features in later releases. It is obvious that the use of modular programming will be very helpful in making this kind of planning decision. This list of responsibilities is not meant to be complete, but does bring out the basic duties of the designers. If they decide to take the modular programming approach, they must find out about all the items listed above during the planning for the design of the logical structure of the whole program, but prior to actually modularizing it.

Many installations using modular programming have guidelines on the size of a module. These restrict the

size of a module by placing an upper limit on the total number of statements it can contain or the total number of possible paths through it. By keeping down the size of modules, the documentation for each is short, simple, and relatively easy to maintain. If a module should need modification in the future, it will be easy to identify where the changes should go and determine how they should be incorporated.

As with all programming methodologies, modular design is not a panacea for every problem area associated with programming. What it does do is to force a better understanding of the problem during the design phase and provide the tools for making more accurate and better estimates. It provides the structural framework for maintaining accurate and understandable documentation. These factors permit management to exercise positive control over the project and get a good overall view of what is happening.

D. Reliability

One of the greatest advantages of modular programming is the reliability of the product that is turned out. This is achieved through testing. The more completely a program is tested, the more reliable it will be since each additional test that is performed reduces the number of untested paths where possible errors could be hidden.

Modular programming helps a great deal in this respect because thorough testing of each module is an achievable goal even though thorough testing of the whole system with all modules integrated together usually is not. Since the modules are largely independent of one another, parallel implementation is possible, and since modules are tested as they are completed, testing is not held off until the end. A programmer's typical work sequence would be to code a module and test it, code another module and test it, etc. This means that as the development cycle nears its end, most testing has been completed except for integration. Integration testing should flush out most of the remaining problems and after this process is finished, the total system should be well debugged and run reliably.

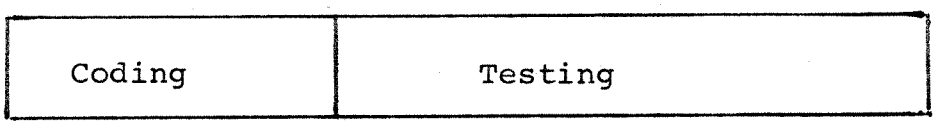
The problem with testing is that there usually is not enough machine time or manpower to test a program exhaustively before the program is needed. The number of paths that a program can take may go up exponentially with each additional branch in the program. A program with 12 branches may have up to 2^{12} or 4096 possible paths within it. An exhaustive test of such a program would consume a large amount of resources, in terms of both manpower and and computer time. If, however, the program were broken into three modules so that each one had four branches, then the number of paths would be reduced to a maximum of

$2^4 + 2^4 + 2^4$ or a total of 48 if each module is considered to be independent of the others. This makes the notion of thoroughly testing each module realistic, but does not alter the fact that after integration, there will still be $2^4 * 2^4 * 2^4$ or 4096 paths to test in the system as a whole. What this does mean, nevertheless, is that testing can be done at the module level to remove almost all bugs and problems from the individual modules. Then the only problems that should arise during integration are those caused by interfaces, linkages, or the passage of data between modules.

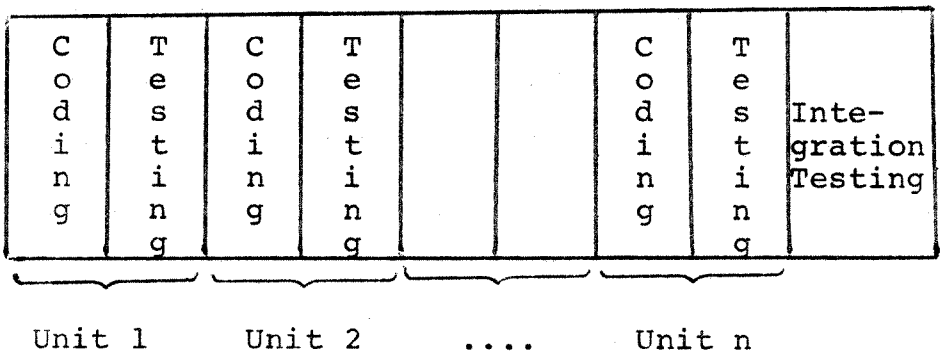
Because modules are small in size and can be coded up relatively quickly, testing starts much earlier in the development cycle than in a non-modular programming environment. This takes the pressure off the programmer for a last minute compression of the test phase to meet schedules and results in better and more thoroughly tested programs. In the conventional environment, all the coding for the entire program is usually done before testing starts. Then, as the schedule slips while the end date remains fixed, the testing period ends up absorbing the loss and shrinking in length. The shorter the testing phase becomes the more hasty and superficial will be the testing which the program receives. (See Figure 1). In the modular programming environment, coding and testing are alternated

from start to finish so that most testing is finished before final integration begins. This insures that the finished product will be in good shape when it enters the user's production environment.

Non-modular program:



Modular program:



IMPLEMENTATION SEQUENCES

Figure 1

A reliable system saves money and time. The more reliable a program is, the less has to be spent in order to maintain it. Usually the amount of money spent on mainte-

nance is a large percentage of the total cost of supporting a program over its lifetime. Any lowering of costs here would represent a significant saving in the total cost, but the actual amount saved is very difficult to determine because the savings are mostly in the form of cost avoidance. Reducing maintenance cost is not the only advantageous fallout from a reliable program, however. There are other benefits such as greater machine availability and a high level of user satisfaction with the program. The user will certainly not be happy with a program that is repeatedly inoperative because of bugs if his people are waiting to use it in order to get their work done. These periods of downtime mean wasted manhours and money lost. Moreover, from the data processing center's point of view, a reliable system is good because it boosts the attitude and morale of the operators and programmers since nobody likes a program that constantly needs fixing and nobody likes to debug under the time pressure of a production environment.

E. Maintenance and Modification

Maintenance is concerned with the continued upkeep of programs. Its main function is to ensure that the programs operate properly and accomplish the functions they were designed to do.

In every data processing environment around 30% to 40% of the total resources are allocated to maintenance. Next to the initial development phase, this activity is the prime contributor to the high cost of software. In most installations, maintenance seems to be an open ended task. Managers in data processing centers must always set aside a large percentage of their total resources to do maintenance. Included in this activity are all the modifications that customers or users of the programs need, whether caused by changes in the specifications, changes in user needs, or because some misinterpretation of the specifications by the system analyst was incorporated into the program.

In order to implement a change, the programmer must understand what the change involves, familiarize himself with the program or programs to be changed, determine how and where to make the alterations, and check carefully to see that the alteration will not have unwanted side effects. Once the change is made, the programmer must run comprehensive tests to check that the modifications occurred as planned and that no unwanted repercussions were induced. The programmer then updates the documentation to add in the changes that he has made.

Modular programming makes the job of maintenance easier. When the program is modularized, the programmer

needs to understand the workings of only the specific module to be altered. He can trace through the modular design plan and locate the module where the change should be made. At this time, he has the option of adding code or rewriting the module in order to incorporate the change. It may be easier to rewrite the whole module rather than add to the existing code because of built-in limitations. However, since modules are small, this can usually be done quickly. He can use the modular design flowchart to determine the linkage from this module to other modules and what, if any effect, the change will have on the other modules.

Once the modifications have been completed, the module needs to be tested. Modular programming permits the use of a phased test plan. Retesting is first performed against the altered module in a standalone situation. Then the module is tested together with those modules that have dependencies on it. Once this has been completed, the module is brought in and tested in the context of the total program. Such a phased test plan insures that the module will not enter the production environment prior to the completion of the first two test phases. As a consequence, most bugs and problems are isolated and corrected outside the main operating environment. It is not until the final test phase, when

the modified module is integrated into the whole program, that the user is exposed to possible failures due to undetected bugs. This protects the user against program failures while allowing the maintenance team to test altered modules without resorting to extreme measures such as restricting testing to third shift machine time.

It is a strong point in favor of modular programming that it allows this phased approach to testing. The use of this approach can at the very least reduce and at the best prevent troublesome system disruptions. Another advantage of modular programming is the relative ease with which documentation can be kept current. Updates to the documentation can be easily tracked and handled because they are needed only for the modules that undergo modifications.

In a modular programming environment, the turnover of personnel is not as critical as in the traditional environment, where one or two 'key' persons carried the knowledge of the entire system in their heads, because documentation was absent or too poorly written to use. The loss of these people could turn the installation upside down. When modular programming is used, however, there should be documentation available for each module in the program. This documentation should be clear and

easy to understand because the modules themselves are small and restricted in scope. In this kind of environment knowledge about the system is retained in the documentation, and not in the heads of a few key people. Also, because it is considerably less complex to gain an understanding of a few modules rather than a whole system, managers can assign regular maintenance functions to junior programmers and be fairly well assured that the job will be done properly. In addition, the maintenance of the program can be done by programmers who were not involved in the original development. This type of staffing is usually necessary since program maintenance is rarely done by the original implementors anyway.

2.5 PERSONNEL REQUIREMENTS

In the majority of installations using modular programming, the staffing level is similar to those not using this technique. Most of the modular teams have a few experienced or senior level people and a large group of lower level or junior people. The senior staff are usually responsible for the overall design of the program and its division into modules. They also oversee the implementation of the modules by the junior level people during the coding, compiling, testing, integration, and

documentation phases.

Programmers opposing this methodology often see themselves reduced to the level of mechanical coders because they feel that the constraints of modular programming rob them of the opportunity to display their creative and technical skills. This complaint is aired most vigorously by programmers who work closely with and know many of the idiosyncrasies and peculiaritites of the machine. These people feel that modular programming takes the fun and technical challenge out of programming. What they fail to see is that well written, reliable, and easy to maintain programs are not built on clever and deft combinations of code. They also fail to see that the career path offered by modular programming is to advance beyond the level of astute coding and into that of program design, an area which takes just as much or more creativity and skill to master.

Training and educating programmers to perform within the framework of modular programming varies in difficulty with the amount of experience the individual has had. Those programmers with less than a year of experience adapt to modular programming with relative ease, while those with longer times in the field, especially with experience on the past generations of com-

puters, have greater difficulty in accepting the use of modular techniques. Their early lack of acceptance stems from reservations that modular programming will limit their creativeness and their usefulness as programmers. These fears are valid in the sense that modular programming places rigid constraints on what a module can contain, what functions it will perform, and how it is to be built and coded. These restrictions are the outcome of a controlled and disciplined approach to program implementation. The really creative processes occur during the design and modularization phase, and this is the area that the programmer should aim for as he gains more experience.

Modular programming techniques have been well received by the managers of relatively young programming teams. They find that within a modular programming environment they can make use of trainees and get productive work from them. The trainees are first taught the basic rules and then are given relatively simple, well defined, non-critical programs to work on. This situation pleases managers because their trainees are producing useful output and are assets to the group. It also pleases the trainees because they have been given a small amount of responsibility and can contribute to the project immediately. If the trainees do well on their initial assign-

ments, they will gradually move upward into more complex and sophisticated jobs. Ultimately, they will grow to the point where they are lead programmers and can handle the design of a whole program and supervise the actions of a team of people.

Installations that have switched to modular programming have found that this technique has boosted their productivity. This increase has permitted groups to handle larger workloads with the same staffing or to reduce the total number of programmers while maintaining the same level of work. These installations have also had positive feedback indicating that their final product was of higher quality than what was previously produced, particularly in the areas of reliability and maintainability.

2.6 INSTALLATION CONSIDERATIONS

A. Size of Installation

There is a widely held belief that modular programming techniques benefit only large programming organizations - e.g., those with over 40 programmers. This belief is based on the premise that larger organizations benefit most from what modular programming has to offer, namely the techniques to:

* Estimate with precision

* Track progress closely
* Handle resource allocation
* Increase program productivity
* Improve program quality
* Implement in parallel
* Spread out machine utilization and program testing

These benefits satisfy many of the needs of large installations and are applicable to the requirements of small installations also. In fact, a single programmer working on a one man project can find it profitable to use modular programming techniques. By applying these techniques while organizing and structuring his program, he can gain certain advantages. These include such things as being able to work on more than one module at a time, being able to make modifications easily, being able to verify that each of his modules works as he progresses, and being able to simplify his documentation effort now and his maintenance efforts later on. Of course, he will not be overly concerned with the close tracking capability, the resource allocation capability, or the tight management control offered by modular programming. Also, in small programming groups, communication among people is simplified so that often there is no overriding need to

impose rigid standards on the formats of programs or on the interfaces between them. Such standards do, however, make it easier to understand and modify the work done by others and, as a consequence, do make sense even for the smaller installations.

On the whole, the personnel in the smaller installations may feel that modular programming imposes unnecessary restrictions and standards on the operation of the group. Such restrictions and standards do, nevertheless, work to the benefit of the group in terms of better documentation, better reliability, and better maintainability of the programs produced. Therefore, smaller installations should not ignore the techniques of modular programming even though they provide some benefits that the small installations do not find useful. The benefits that remain should still be of considerable interest to these installations.

B. Type of Installation

The use of modular programming techniques benefits those installations that have high maintenance loads and large amounts of development work in progress. During development and implementation, these installations can take advantage of the modular techniques that result in more straightforward and more effective program mainte-

nance. The improvement in maintainability that modularity provides could result in considerable savings to the installation over the useful life of the program. If, however, the installation is in a non-modular programming environment, what should be done? Those installations which do a large amount of development work in addition to maintenance should adopt a policy of modularizing all their new work. Those which do very little development work and whose function is primarily maintenance will have no impetus to convert to modular programming since the cost of modularizing existing programs could not be justified. The only exception would be a case in which the anticipated life of the program is long and its present condition is so bad that a new version of the program appears to be a reasonable solution to current maintenance problems.

Today, there are more scientific than commercial organizations using modular programming. Undoubtedly, this has occurred for several reasons:

(1) Scientific problems are much more complex than commercial problems and are more easily handled when broken up into smaller units.

(2) Scientific problems lend themselves more readily to modularization because they are more functional in nature.

(3) Scientific problems contain functions that are likely to repeat so that partitioning them into reusable modules is logical and practical.

Frequently, the modules written to solve scientific problems are grouped together into a library which can be referenced, for example, during a mathematical calculation, a statistics table lookup, or any process oriented function. The scientific community has had a lot of experience with modular programming and a great deal of influence over its development. With the ongoing increase in the complexity of the applications handled by commercial programming, this area will certainly move more and more in the direction of modular programming to take advantage of the disciplines that it offers.

2.7 SUMMARY

Installations that have used modular programming have experienced the benefits that this programming technique provides. These benefits can be grouped into five broad categories.

(1) Better project control through more accurate estimates and more even distribution of workload. Because the overall program is divided into small

modules or work units, more precise estimates can be made for the coding, testing, integration, and documentation phases of development. During implementation, management can track the progress of the project by monitoring the progress of each module and measuring it against its own individual schedule. Differences between the actual progress and the estimated progress in any one area can signal adjustments to the workload immediately. The size of the difference will determine the severity of the adjustment.

Project control is also enhanced by the independence of the modules with respect to one another. This characteristic allows programmers to work on different modules in parallel. This promotes a more even distribution of workload on the people and on the machine as well.

(2) Better program design and better utilization of the code. The design of the program is usually in the hands of one or more of the senior level programmers. They design the modules only after careful evaluation and study of the user's requirements and specifications. Their design includes the data interface and the dataflow between the modules and also identifies

any common or reusable modules.

(3) More reliable programs through more complete testing. Modularity facilitates testing because small modules are easier to test and test comprehensively than large groups of code. If testing was done thoroughly at the module level, the only errors that should occur during the integration phase are those that are related to the data flow or data synchronization between modules.

(4) Easier maintenance through modularity and clear documentation. Since each module in the program performs a well defined function, any single modification will be limited to one or only a few modules. This localizes the scope of the change in terms of what the programmer who is performing the maintenance has to learn and understand and also in terms of its effects on the rest of the program.

(5) Better resource allocation. With the program divided into modules, management can assign tasks to the programmers based on their abilities. Trainees and inexperienced programmers can be given simple and well defined modules, and the experienced programmers can be given the more complex modules. This arrangement allows for the training of new programmers while they

are contibuting to the job.

Accrual of these benefits will come to the installation only if it is strongly committed to modular programming. This programming technique must have management support to put it into operation. It cannot reach full effectiveness unless the proper environment is created for it. This generally means providing the programming staff with a substantial amount of education and training in modular programming concepts and methods. It also means working out and establishing a reasonable set of programming standards and restrictions for the installation. The transition period will involve a lot of work, and management will probably pass through some trying times before it can begin to obtain the benefits of modular programming. Once the conversion has been completed, however, the benefits derived from modular programming should convince management that it was well worth the effort to switch over.

2.8 REFERENCES

1. Canning, R. (ed) The Search for Software Reliability. EDP Analyzer, May 1974.

This paper gives the basics of modular programming and describes how different installations have moved into using this technique. It is a good source for

references.

2. Hoskyns Systems Research, Inc. Implications of Using Modular Programming. Central Computer Agency, Guide No. 1. New York, N.Y. 1973.

   This document has one of the best bibliographies on modular programming. It contains a great deal of material and covers this programming technique in depth. Items included are modular programming characteristics, disadvantages, pitfalls, installation requirements, and more.

3. Myers, G. Composite Design: The Design of Modular Programs. IBM Corporation, Poughkeepsie, N.Y. 12601. TR00.2406. 1973.

   This report provides a complete set of modular programming specifications. It includes descriptive samples and illustrations to help orient a potential user of modular programming.

4. Parnas, D. A Technique for Software Module Specification With Example. Comm. ACM, May 1972, Vol. 15, No. 5.

   This document is a complete and precise guide to modular programming.

5. Parnas, D. On the Criteria to be Used in Decomposing Systems Into Modules. Comm. ACM, December 1972, Vol.

15, No. 12.

This article describes ways to use modularity in programming. It is a good reference document for the principles of modular programming.

CHAPTER 3

THE CHIEF PROGRAMMER TEAM

3.1 INTRODUCTION

A new concept in programming methodology was introduced by IBM in the early 1970's to deal with the technical as well as the managerial problems associated with the development and construction of large systems. This concept was based on restructuring the programming group into a closely knit team organization called the Chief Programmer Team. This organization was structured so that its members would have specialized and well defined duties. Heading the group was the chief programmer who functioned as the operational manager and also provided the technical leadership for the rest of the team members. He was assisted in his duties by the backup programmer, the programming librarian, and a small staff of high level and experienced programmers.

Up to this time, most programming groups had been staffed by relatively inexperienced or junior level people. Because of their inexperience, much of the work done by

these people in the design, coding, testing, and documentation of programs had been unacceptably low in quality. The reliability and maintainability of their programs was well below par and was the cause for many complaints. The Chief Programmer Team concept was developed in an endeavor to meet these problems head on. Included in its fundamental tenets was the notion of reintroducing senior level people back into the detailed programming environment. This would put them back onto the front lines, so to speak, to participate in all the action of development and in a position to use their experience and skill to guide and influence program design and to help overcome any troublesome stumbling blocks. The Chief Programmer Team concept has attempted to attack the technical problems of program development from a position of strength. This position of making the best people available for handling such problems is reinforced in many cases by the use of structured programming and top down programming, two concepts which will be discussed in the next chapter. The merging together of these three factors results in a formidable combination for dealing with the general kinds of problems encountered in the development and implementation of programs.

3.2 TEAM ORGANIZATION

The Chief Programmer Team is a highly structured group made up of skilled and experienced professionals, each with one or more specific functional roles to fill. The nucleus of a Chief Programmer Team consists of the chief programmer, the backup programmer, and the programming librarian. This small central group is supplemented by a staff of specialists including system analysts, programmers, programming technicians, and technical writers. The take their direction from the chief programmer and help wherever they are needed. Any administrative problems and all personnel related duties are handled by a project manager, thus freeing the chief programmer to concentrate on the technical aspects of the project.

A. Chief Programmer

The chief programmer is the key figure in the Chief Programmer Team. In most cases, he is a senior programmer or analyst who has shown that he is capable of designing and building a major programming system. Overall, he can be characterized as an individual with proven organizational and leadership abilities, a strong technical background, and a substantial amount of programming experience. He possesses a good working knowledge of

many programming techniques and is highly competent in system analysis and design. He is well known for the depth of his technical expertise and more than likely has acquired the reputation of being a programming problem-solver.

For his role as chief programmer, he is the team leader and functional manager of the group. As technical leader, he establishes and provides the technical direction for all the other members of the team. He maintains a close working relationship with the backup programmer and confers or consults with him over most important technical matters. As chief programmer, he assumes full responsibility for the entire system. This makes him the prime architect for the system, and he plans the design, implementation, and testing phases with the help of the backup programmer. After the planning phase is finished, the two of them generally work on the design of the high level portions of the system and let the system analysts and programmers work out the lower level portions under their supervision. Later, during the implementation stage, they assign and direct the generation of code for numerous programs and routines that make up the new system. When the coding phase is completed, the chief programmer and the backup programmer review and oversee the testing and