Abstract

In recent years, as the hardware cost/capability ratio has continued to decrease and as much of the routine data processing has been computerized, the emphasis in software development has shifted from just getting systems operational to the maintenance of existing systems, reduction of duplication by integration, selective addition of new applications, systems that are more usable, maintainable, portable and reliable and to improving the productivity of software developers.

This paper examines a number of trends that are changing the methods by which software is being produced and used. More and more of the research and development is now being directed towards producing systems that have the desirable properties mentioned above. Also, more computer-aided tools are being developed and made available. The most important trend probably is the introduction of software development support facilities which provide an integrated set of tools, based on a central computerized data base. At this stage, these systems perform primarily clerical work but gradually their facilities are being expanded. A number of research and development efforts are concerned with moving closer to the ultimate objective of producing executable software for a particular computing environment directly from a set of functional requirements.

A large part of the software developed today is still custom built to meet a specific set of requirements in a specific computing environment. Considerable effort is being directed towards reducing the amount of software that must be newly produced each time by making use of software that already exists.

Many of the concepts and techniques in which these developments are based are not new. Many tools are available but are not widely used. The availability of technology does not necessarily mean that it will be used. Some of the reasons for this are examined as a basis for the consideration of issues involved in practical application of new system development technology.

1. Software Environment

1.1 Computer Based Information Processing Systems and Related Terminology

Large amounts of resources are being devoted to development and maintenance of computer based information processing systems, i.e., systems that include hardware, software and other components. Much of the concern with the efficiency and effectiveness of these systems today is focused on the software e.g., "software is the problem". This paper is primarily concerned with software development, however it is desirable to first place software in its proper context within systems. Terminology is not yet standard in this field, and it is therefore necessary to start with a number of definitions. In general these are consistent with those used in the recent encyclopedias by Balston and Neek (1975) and Belzer, Holzman, and Kent (1975).

1.1.1 Classification of Systems. The terms such as systems, information systems, information processing systems are used with widely different meanings. The meanings in which they are used in this paper are defined in this section and illustrated in the example in Figure 1.1.1.

Organization

An organization is a legal entity, or sub-unit of a legal entity, that is uniquely and specifically identifiable. Usually an Information System is a part of an organization which has a basic function other than information processing. In Figure 1.1.1 the example of an organization is a Manufacturing Enterprise.

Organizational sub-systems

An organization may have a number of sub-systems which are used to accomplish, or contribute to the accomplishment, of the basic function. One of these subsystems is an Information System. In Figure 1.1.1, the Manufacturing Enterprise has sub-systems such as Production, Logistics, Distribution, Finance, etc.

Information Systems

The Information System is the sub-system of an organization in which information (in the form of data) is received, recorded, processed, stored, retrieved and transmitted. The Information System may consist of an informal system and (formal) Information Processing Systems. The informal system consists of essentially information processing in which data is not recorded.

Information Processing Systems

The subsystems of the Information System in which data are recorded and processed following a formal procedure are called Information Processing Systems. Two kinds of Information Processing Systems may be distinguished: manual and computer-based. Manual systems are those in which all operations are performed manually. Computer-based information processing systems are those in which some operations (though not necessarily all) are performed by a computer.

Software System. A software system is a collection of software components that, if used together, accomplish a complete information processing function.

Software Subsystem. A software subsystem is a part of software system that, if used together,
accomplishes a separately defined part of the whole software system's function. In very complex software systems it may be useful to distinguish more than one level of software subsystems.

Software Components. Software components are parts of software system that constitute basic individual units such as programs, routines, subroutines, modules, units of data description and executable modules.

Software Module. The smallest part of a software system that can be utilized by a software component at the same or higher level to accomplish a defined function.

Statement. The smallest self-contained component of a Software System which can be expressed in the source programming language.

In a particular system, these definitions can be made more specific since they depend on the operating system under which the system is to be run and the programming language in which it is written. Since the discussion in this paper is independent of operating systems and programming languages, the terms system, subsystem module, and statement will be used with the general meaning given above.

Software may be classified by its operational status. An excellent taxonomy is given by Brooks (1975) Figure 1.1.1:

"In the upper left of Fig. 1.1.2 is a program. It is complete in itself, ready to be run by the author on the system on which it was developed...

There are two ways a program can be converted into a more useful, but more costly, object. These two ways are represented by the boundaries in the diagram.

Moving down across the horizontal boundary, a program becomes a programming product. This is a program that can be run, tested, repaired, and extended by anybody. It is usable in many operating environments, for many sets of data...

Promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Moving across the vertical boundary, a program becomes a component in a programming system. This is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks...

A programming system component costs at least three times as much as a stand-alone program of the same function. The cost may be greater if the system has many components.

In the lower right-hand corner of Fig. 1.1.2 stands the programming systems product. This differs from the simple program in all of the above ways. It costs nine times as much. But it is the truly useful object, the intended product of most system programming efforts."

Software is frequently classified as either system software or application software. System Software is used during the execution of other software; it includes operating systems, monitors, etc. It is necessary because the application software cannot run on the raw hardware itself.

Application Software includes software components tailored to the user's needs, including all software products that are not part of system software as defined above.

The application area that is of most concern in this paper is that of software development. All programs supporting the development of software, such as assemblers, compilers, translators, program generators, design-aid packages and the like are included in this classification.

It will be convenient to distinguish between generalized software and custom-tailored software. Generalized Software is software produced to meet the needs of a number of users for the same general application in different organizations and different computing environments. Custom-tailored Software is software designed to meet specific requirements in a specific situation and for a specific environment.

1.2 Environment in Which Software is Developed and Used

Since software is developed in many different circumstances and used in many different situations, it is desirable to classify the different cases and to attempt to identify some parameters which characterize them. Boehm (1976) p. 1239, considers the application of software engineering to two areas:

AREA 1: Detailed design and coding of system software by experts in a relatively economics-independent context.

AREA 2: Requirements analysis, design, test and maintenance of applications software by technicians in an economics-driven context.

The parameters that he considers in distinguishing between Areas 1 and 2 are:

- whether the requirements analysis is done before or after the design and coding start
- whether the software is "system" or "application" software
- whether the development is done by software experts or by technician programmers
- whether (economic) efficiency is relevant or not.

However, he does not consider other parameters which are also important. One of these is the

--------------------

1System software is usually considered as being provided by the hardware manufacturer though Dolotta, et al. (1976) predicts a separation between the user's Installation Control Program (ICP) and the vendor-supplied System Control Program (SCP).
motivation which causes the software to be produced. There appear to be at least the following reasons for organizations to produce software: for their own use, for other organizations under contract for sale (or lease) to other organizations, to sell a service, to sell hardware, or to carry out research.

Another important parameter is the way in which an organization meets its requirements: by buying a service, by having a software system developed by others under contract, by purchasing or leasing available software, or by developing its own software.

The combination of these and other parameters leads to the identification of six major environments in which the software problem is important.

The Systems Department in an Organization. The unit in an organization that runs the production operations which produces results for various users in the organization is frequently called the Systems Department. Its production facilities include computers, systems software, applications software, data bases, etc. The department maintains the existing application software and also develops new systems. Many new systems involve integration of existing systems. The Systems Department may develop new systems, have systems built under contract or acquire existing systems.

Organizations which acquire software systems by contract. This environment is similar to the previous one in that the software must be maintained. However since the organization did not develop the software itself, it must depend on the developer for the documentation that is necessary for use and maintenance.

Software Enterprises. These are organizations that produce software for others either under contract or for sale as standard products.

Companies using software to offer a service. Such business services are becoming more common and are predicted to increase, e.g., Seidman (1975):

"The line between computer services vendors and computer equipment vendors is beginning to blur. By 1980, most organizations will be buying solutions to problems instead of buying hardware and/or services. In order to deal with the rapidly increasing competition at both the high and low revenue ends of the spectrum, computer services vendors will evolve into Multi-Service Vendors (MSV's), selling solutions to users' problems. This will include, as required, the provision of hardware, software, and facilities management (FM), in addition to the traditional remote computing and batch services currently offered. To the dp user, this will mean a greater opportunity to get the most efficient and most economical solution to his problem with a minimum of comparison studies and installation and integration problems."

Hardware Manufacturers. These produce system software, particularly operating systems, and other software that interfaces directly with hardware.

Academic, research and development organizations.

These produce software primarily to demonstrate feasibility rather than to accomplish applications in a production environment. State-of-the-art software is produced by experts primarily for research purposes. In this situation, the software is not intended to be used for operational uses, and cost and performance may not be as important as demonstrating feasibility.

The above environments are ones in which a large amount of software must be produced and maintained over a long period of time. During this time the requirements and computing environments may change. In such an environment any given software system is difficult to maintain. Further, since the software is always being changed, it may exist in many versions, each with many releases.

1.3 System Development, Software Development and Software Engineering

1.3.1 Evolution of System Development Methods. Software development is only one aspect of Information Processing Systems. It is, therefore, desirable to examine the evolution of system development because improvements in software development must be compatible with the changes in the total environment.

Initially systems were developed by ad hoc methods. Such an approach is practical only for very small systems where only few people are involved. Since it is usually not feasible to divide systems into separate, distinct small systems, this approach is not relevant to the environments described in section 1.2.

The traditional or classical approach to system development in the System Life Cycle method based on a sequence of phases controlled by a project management system. The approach, covered in many books and publications, (e.g., Hartman (1968), Hice, et al. (1974), Metzger (1973)), consists of: (1) specifying a number of activities that must be carried out, (2) identifying a number of phases with the results to be obtained in each, and (3) describing the results, i.e. documentation, which must be produced at the end of each phase. Figure 1.3.1, taken from the front cover of Metzger, illustrates these aspects that characterize all such methods. Frequently, the procedures are divided into a particular type of system. For example, the U.S. Department of the Navy (1974) has a documentation standard for tactical digital systems.

Even though information systems are a type of system, few applications of systems theory have appeared in the literature. One interesting example is the attempt to apply control theory to information systems by Lehman (1969), Belady and Lehman (1971), and Belady and Lehman (1976). Their models of the life cycle of systems include the concepts of releases and versions. Another example is given by Putman (1977) who analyzes a Systems Department with many systems in different stages of their life cycle.

1.3.2 Software Development. The System Life Cycle method usually includes the following activities in software development:

- Determination of the requirements that the software is to accomplish
- Design of the software including specification of the major modules
- Design of the individual modules

- 191 -
The exact procedure that has to be followed in a particular computing environment depends on the methods used for design, the conventions and standards that have been adopted, the programming language, and the operating system.

1.3.3 Software Engineering and Software Production. While the System Life Cycle method is in widespread use, the model on which it is based is too simple for many practical situations today. One improvement which has been proposed is frequently referred to as Software Engineering. The term has been defined by Boehm (1976) as follows:

"SOFTWARE ENGINEERING: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them."

The Software Engineering concept implies a change from preoccupation with programming to a broader view encompassing software (i.e., a collection of interrelated programs), recognition of the need for trade-offs among a number of criteria and consideration of the applicability of engineering practices. However, the term still is too narrow for a comprehensive view of systems since software is only one part of systems; systems also include non-computerized procedures and hardware. The non-software or manual procedures include many aspects of user interfaces that determine the value of the system to the organization which is paying for it.

In this paper, the term "Software Engineering" will be used to describe the initial design and development of prototype software. The term "Software Production" will be used to describe all the activities necessary to produce software that is satisfactory for operational use.

1.3.4 Personnel Classification. An important consideration in improvement in software development is the personnel associated with systems and their respective roles. Four major groups of personnel involved with can be distinguished: the software personnel, the information processing personnel, the personnel using the target system and the organization's management. The software personnel are directly involved with the software and are concerned with its development, operation and maintenance. These are Software System Designers, Programmers, the software-hardware Operators and Program Librarians. The second group - the information processing personnel are concerned with the developments, operation and maintenance of the entire organization's information processing system. These consist of Systems Managers, Project Leaders, System Analysts, System Architects, Data Base Designers, Data Base Administrators, System Operators, Hardware Engineers, Communication Engineers, Project and System Librarians. The third group, the target system's users, consist of organization's personnel and its customers or customers' personnel. This group is involved with preparing input data, observing the software format and using the output data to perform its tasks. The fourth group, the organization's management, consists of persons bearing the overall responsibility for the organization's image and the organization's economic effectiveness. Among this group are the managers responsible for the Systems Department in the organization.

1.4 Software Costs and Other Characteristics

While it is clearly desirable to "improve" software, in the software field there is as yet no satisfactory way to measure progress in quality or productivity. Currently, the most common measure is lines of code. Even the basic definition of "code" is subject to interpretation, e.g., does it include comment lines? Furthermore, the use of this measure as a productivity measure leads to undesirable results. If a programmer knows that he will be judged by the number of lines of code he will be motivated to avoid the use of macros and libraries, to use low level programming rather than higher level statements, and avoid generalizing code into reusable units because this reduces the number of lines of code and also tends to make the resulting code more difficult to produce. The lack of satisfactory quantitative and scientific methods and adequate measures is emphasized in most of the surveys of the state of the art of information systems. See, for example, Dolotta et al. (1976).

There are also no generally accepted characteristics in which improvements can be measured. Terms such as reliability, quality, maintainability, etc. are used without definition or with different definitions by different authors. Recently, several efforts have been made to define software characteristics and develop a method for measuring the values of the characteristics for a particular unit of software. See, for example, Boehm (1973), Gilb (1976) and Reifer (1976). The characteristics tree developed by Boehm is shown in Figure 1.4.1. Here characteristics will be divided into two categories: cost and other. The "other" characteristics will be divided as in Telchrow (1974).

1.4.1 Software Cost. Three methods for reducing the cost of software to the final user may be considered:

- Reducing the cost to develop software to meet given requirements. This in turn can be accomplished by improving the productivity of software developers or by having more of the software developed by machines at lower cost.

- Reducing the amount of new software which must be developed to meet a given requirement.

- Producing software which will be used by a number of users. The cost to each should decrease proportionately to the number of users. In practice the decrease will not be proportionate since some of the cost such as maintenance increases as the number of users increases. Furthermore, obtaining a number of users to use the same software inevitably requires a marketing effort.

One of the difficulties in improving software is deciding what should be included in the cost of software. It is now being recognized that the
cost of a system is much more than the cost of development. For example, there are costs involved in training and documentation. It is also being recognized that development costs may be small relative to maintenance costs during the life of the system. Decisions on systems are frequently based on cost of software only, and decisions on software are frequently based on development costs only.

1.4.2 Software Characteristics. Ideally software should have three major types of characteristics: accomplishing a given set of requirements, be efficient in its use of resources, be easily changeable as the situation changes.

Achievement of these goals usually involves a tradeoff since improvement in one may result in degradation in others. Systems are built to serve organizations. Consequently the systems must accomplish the requirements that are stated by the organization. A first objective, therefore, is to build systems that accomplish those requirements. Requirements are seldom absolute; the organization and its environment change and therefore the requirements may change. Also, requirements are not independent of cost. There is usually a point beyond which it is not worthwhile to achieve the requirements. Thus, it is necessary to decide when the requirements and the cost of a system decreases the organization becomes more willing to adapt itself to the system rather than insisting on requirements which may cost more.

Many of the techniques considered in this paper have a potential of reducing the need for the tradeoff by making it easier to change software to fit a particular situation.

1.5 Improvement of Software Development

The attempts to improve software development arise out of the recognition of the need for changes and the recognition that improvement is possible.

1.5.1 The Need for Improvement. A number of studies of the state of software development have shown that the quantity and quality of software being produced is already a limitation on effective utilization of computers and that this problem will become even more serious in the future. Two studies were conducted for the U.S. Air Force (CCIF-75 and SADPR-75); the first has been summarized by Boehm (1973). An extensive study has been conducted by the SILT committee of SHARE, Dolotta et al. (1975). IBM has conducted a survey on application development problems (Perry 1975). A general overview of the effectiveness and efficiency of System Departments is given by Canning (1975). While these studies contain some quantitative data, comprehensive data is hard to obtain. (An excellent summary of available data is given by Phister (1975).) Development problems sometimes have to be reported as hypothetical situations as in the MUDD report (Welsh 1975). One situation where quantitative data sometimes appears is in situations which have to be resolved by legal methods. (Wiseman 1976).

There seems to be no question that improvement in qualitative characteristics is essential. Furthermore the amount of software needed in the future cannot be produced by present manual methods (Prywes 1975 and Dolotta 1976).

1.5.2 Possibilities for Improvement.

Industries which start with a new product or service go through a life cycle. In the initial growth phase, the industry is rapidly expanding its markets, frequently by expanding the capability of its product or service. The industry reaches maturity when its product can no longer find new markets; it is then limited to the replacement market. If that market shrinks enough, possibly due to newer products, the industry becomes extinct.

The computer industry has been evolving and clearly is still in a growth phase. The rate of evolution of computers has been very rapid. Evolution of computers is commonly divided into "generations", each of which centers around a major jump in capability and/or decrease in cost. The generation concept has been applied to hardware, (Withington 1974), operating systems, (Denning and Nolan 1974), systems development (Benjamin 1972), and systems analysis techniques, (Couger 1973, 1974). The evolution is still continuing, particularly in hardware. The cost of hardware to perform a given computation has been decreasing and the relative cost of hardware to software, in a given system, has also been decreasing.

Since hardware costs have been decreasing it is worthwhile to examine the reasons and to see whether the same approach could be applied to software to reduce the cost of software and at the same time improve its quality.

Hardware costs have decreased for three major reasons:

- Research in the underlying phenomena. This resulted in discovery of ways of performing basic operations at much higher speed with much greater reliability.

- Differentiation of functions involved in the hardware system life cycle, and development and use of appropriate mechanized tools for each of the functions. The separate functions that exist in most hardware manufactures are: engineering, prototype development, manufacturing, marketing, and maintenance.

- Mass production of components and sub-assemblies. This has permitted investment in production facilities to reduce the per-unit cost.

The analogy with software production is not perfect and some of the relative costs are different. For example, the cost to produce another copy of a program is insignificant whereas the cost to produce, for example, another disc drive is not. Nevertheless if one includes all relevant costs associated with information processing systems the analogy becomes much closer.

This paper is concerned with the application of the concepts in the last two points above (differentiation of function, and mass production) to software development. In particular Section 2, 3 and 4 describe the tools that might be used in each of the functions in the software system life cycle. Section 5 covers the software analogue of mass production; namely the use of existing software components. Section 6 discusses the technical, operational and economic feasibility of changing the development of software. Any approach to improving productivity should take
into account the evolution occurring in the computer field. On one hand, the type of software that is to be developed and the manner by which software productivity is measured will be affected by evolution in hardware. On the other hand, the investment in software at one stage in the cycle may limit the opportunity to make use of new hardware capabilities at another stage.

One factor affecting productivity is the type of systems being built. Systems are becoming large and more complex. Consequently, software development emphasis is on the sharing of facilities. However, the increase in the availability of mini-computers has led to distributed processing which may both make systems smaller and simpler and resource sharing less important. One example of the possible changes in software development is indicated by Rigo (1977):

"We have all heard about Citibank's massive conversion to minicomputers. We are now beginning to hear the rest of the story and it involves a lot more than hardware. In somewhat oversimplified form the current situation was recently described like this:

* Citibank, one of the world's largest corporations, is junking all its big IBM mainframes.
* It is replacing its big machines with minis - dozens of them, of every make and model.
* It has drastically reduced its permanent systems and programming staff. For new development, it rents consultants for as long or short a time as needed.
* It used to take two to five years to design and implement a new system. Now the bank is getting the job done in as little as six weeks."

1.6 Summary and Conclusions

In the development of software that is to be used operationally in computer based information processing systems (see definition in section 1.1), the emphasis must be on "programming system product" as specified by Brooks (1975), Figure 1.1.2.

The primary use of programming system products is in System Departments as defined in Section 1.2. The environment in which these techniques are most relevant is one in which a large amount of software must be produced and maintained over a long period of time during which the requirements and computing environment may change. Any given software system in such an environment is large with hundreds of modules and tens of thousands of lines of source statement. The software must be developed and maintained by staff whose membership is also changing. The software may exist in many versions each with many releases. Techniques that are useful in this environment are not appropriate for situations in which state-of-art software is produced by software experts primarily for demonstration purposes. In these situations software is not intended to be used for operational uses and does not have to be maintained and cost and performance may not be as important as demonstrating feasibility.

Software may be developed by the using organization, by software companies under contract, by software companies for sale or lease or by research institutes. The techniques to be discussed are appropriate to all except the latter.

Currently much of the software intended for operational use is produced under the system life cycle approach as described in Section 1.3. This approach has been characterized by the following:

- most software is developed manually with little use of tools other than compilers, editors, linkers and loaders, and operating systems.
- most software has been designed to satisfy a particular requirement in a particular computer environment.

To improve software development it is necessary to be able to identify the characteristics in which improvement is desired and to be able to measure improvement. These characteristics are examined in section 1.4. The most important are:

- to reduce the cost of software to a particular user
- to improve the degree to which the functional capabilities meet the users needs
- to increase the likelihood that the software functions correctly and efficiently.

A basic premise is that software development and maintenance will change in the following major ways:

- software development in the future will be aided by coordinated set of tools (manual as well as computer aided).
- more of the software development will be done by computers.
- the amount of new software, that would otherwise have to be produced, will be reduced by synthesizing systems out of existing software modules.

Interest in improvement originates with the desire to utilize more fully the potential capability of computer and this is pursued because there are indications that improvement is possible, as indicated in section 1.5.

The basic motivation for the approach to software development outlined in this paper is that a substantial reduction can be obtained in the cost of a software system to an individual user while at the same time making it more reliable and more adaptable to changes in the computing environment. However, these new approaches will not be adopted automatically. Some of the reasons why progress has not been faster are examined in section 6.

These considerations lead to the following conclusions:

- The productivity of software developers should be improved by making use of the computer for data processing activities.
- The productivity of software developer can be further improved by automating as much of the software development and
maintenance as possible.

- The productivity can also be further improved by reducing the amount of new software to be developed.

A survey of the state of the art of techniques to accomplish these approaches to improvement of productivity is given to provide the specification for a Software Support System. The current status of development along these lines is summarized in the following sections. The tools available for information processing system development are surveyed and several state of the art approaches to integrating the tools are examined in section 2. The specification for a software system capable of supporting information processing systems during their entire life cycle (development, operation and maintenance) is outlined in section 3. Section 4 describes approaches to using the computer to develop software. Section 5 is concerned with reducing the amount of new software that must be developed.

2. Tools for Software Development, Operation and Maintenance

The tools for system development, operation and maintenance may be divided into manual tools and computer-aided tools. The manual tools are discussed in section 2.1. Computer-aided tools are classified and some of the factors that limit their use are discussed in section 2.2. The next four subsections describe briefly some attempts to overcome these limitations: the National Software Works in section 2.3, the System Development Laboratories in Section 2.4, the Application Management System in section 2.5 and the Information Automat in section 2.6. Some conclusions on the use of tools are given in Section 2.7.

2.1 Manual Tools

The term "manual tools" is intended to cover all the skills, knowledge, disciplines and procedures that are used in system development, operation, and maintenance. To a large extent these tools have been developed out of necessity in order to use the computing hardware and software as they have evolved. The tools therefore, are ad hoc and in many cases have to be learned by experience since they have not been adequately documented.

In the last few years considerable attention has been directed to the Improved Programming Techniques developed by IBM. These include Chief Programmer Team organization, Structured Walk Through, Program Librarian, Top Down Development and Structured Programming. These techniques achieved considerable attention as a result of the spectacular results reported by Baker (1972) when they were used in a project for the New York Times. Consequently many organizations have adopted these methods and consider them worthwhile.

However, it should be noted that these methods by themselves do not necessarily lead to software that has all the desirable qualities. This is indicated by the experience of the New York Times with the system produced by IBM, as reported by Flauger (1976):

- Number of errors. "The developers claimed, in fact, a record of one bug for every 10,000 lines of delivered code:

That's all they found because they didn't do that much testing. We figure that if there's only one bug per 10,000 lines of code, then IBM owes us about a million additional lines."

- Hard to maintain. "Why was it hard to maintain? Among other things it was poorly parameterized. When the staff attempted to introduce a new type of display terminal, they were horrified to discover that all sorts of modules 'knew' how many lines there were to a page, for instance, and how many characters per line. It was a major effort to make the code more general."

"The major drawback is that three or four experts are still required to maintain the code. Ms. Szuch estimates that it takes a newcomer about six months to get on board and safely make changes."

- Efficiency. "It was slow, for one thing, too slow to be marketable. And when the maintenance programmers looked inside to see how to tune it up, they weren't terribly happy about what they found. The upshot was that the system limped along for about nine months with only one 'friendly user' (i.e., non-paying), while The New York Times programmers puzzled out the code and hammered it into decent shape. The Data Bank is alive and working today, but it has yet to fulfill its original promise."

- Quality of code. "As for the quality of the code, the Times programmers found it to be highly variable. They soon learned to spot the individual writing styles of different member of the development team. 'You could hardly call it "egoless" code'."

- Design decisions. "But she is critical of some of the design decisions made which, she says, 'meet the letter of the stated requirements but not the spirit.' She suggests that computer professionals have a duty to inform users of specifications that will be regretted later."

This particular example, of course, does not mean that these Improved Programming Productivity techniques should not be used. In fact, practical experience with these techniques in many organizations suggests that they should be adopted, but the desired benefits will not be obtained unless the means to achieve them are explicitly included.

One of the reasons for the lack of adequate manual tools is that there has as yet been relatively little progress in the investigation of the underlying phenomena of information systems; there is, in fact, little agreement as to what classes of phenomena are really basic. Wegner (1976), for example, considers

"the ramifications of four influential definitions of computer science:

2. Computer science is the study of..."
He concludes that

"these definitions reflect four different approaches to the study of computer science and has given rise to four very different paradigms for computer science research. Computer science is perhaps unique in the diversity of its admissible research paradigms, and this diversity is perhaps necessary in deciding how computer scientists should be trained and for differences of opinion concerning the nature of computer science research."

2.2 Computer-aided Tools

Many computer-aided tools for system development, operation and maintenance have been produced. In particular, there are the compilers, linkers, loaders, operating systems, utilities such as sort, which exist in every computing installation of any size. There are also many other tools intended to aid the development of application software.

Each computing facility contains usually only one operating system and only one, or very few, compilers. The use of these tools is essential for use of the facility and therefore controlled and enforced. Since the entire operation of the computer facility and portfolio of application software depends on these tools, there is a great reluctance to change the basic system software or even to allow any way to do it. (Unfortunately, either the use of application software development tools frequently is incompatible with the existing system software, or the benefits come from overcoming the deficiencies of the system software, and therefore, a basic conflict arises. The problem becomes one of obtaining the benefits from the application software while at the same time maintaining stability in the system software. (The virtual machine concept is one approach to overcoming this dilemma.) Directories of tools to aid in the design, development, operation, modification and maintenance of information processing systems, have been published, e.g., by ICP and Datapro. More detailed descriptions have also appeared. For example, Naftaly, et al. (1972) have described a number of tools available to aid development of systems in COBOL; Reifer (1975b) has provided a list of tools available in the U.S. Air Force. Several authors have attempted to classify available tools: Reifer (1975a) and Curry (1976). Reifer has developed definitions for the 60 types of tools; the types are listed in Appendix 1.

All evidence clearly indicates that despite the existence of many computer-aided tools, very few are being used and most of these are not used very extensively. This conclusion was the opinion of many at Session L351 of the recent SHARE XLVII meeting which was devoted to a "Survey of Facilities for Application Development." The session report contains the following evaluation:

"There are hundreds of tools available to aid the application development process. Some installations have over a hundred separate products installed but find that most are used by only a handful of people and many are used only once or twice. It appears that there are some problems associated with the use of the tools presently available: 1) poor human factors, 2) lack of education and publicity, 3) incompatibilities with installation standards and other tools, 4) not a part of an integrated system, 5) specialized nature of some tools (i.e., only good at serving one particular non-recurring problem) and 6) unsatisfactory performance (i.e., does not improve productivity of the application development process). In addition, most users are not aware of the range of tools available, their benefits, and other users experience with them."

The major reasons for the limited usage of tools are:

- Knowledge of existence. Despite the attempt to provide directories of existing tools, it is frequently difficult to determine where a tool with the required characteristics for a given task exists.

- Access and availability. A tool may not be available to a person, even if he knows of its existence, understands it and wishes to use it, because he does not have access to the computer system on which it is installed.

- New languages. In order to use a tool, it is usually necessary for a person to learn two languages. First is the language used by the tool itself and second is the operating system or command language. The first is almost always a new language while the second may or may not be.

- Interfaces. Most tools have their own input and output formats and consequently, each user must reformat the data to be acceptable to the tool and then reformat the output if he needs it for another tool.

- Status. Many tools, in the terminology used by Brooks (1975), are "Programs" rather than "Programming Systems Products". They have inadequate user documentation and are inadequately tested. After a certain amount of trying to use a tool without success, the user may well give up and decide the effort to get the tool working is not worth it.

The potential benefit of using tools in systems development has lead to a number of attempts to overcome these difficulties. These approaches are characterized here by the degree to which they make use of what already exists. The first approach starts with the premise that it must be possible to incorporate existing tools essentially unchanged. An example is the National Software Works (Section 2.3). The second approach also makes as much use of existing tools as possible but attempts to integrate them by providing common interfaces. Examples are the Software Development Laboratories described in Section 2.4. A somewhat similar approach is incorporated in the Application Management System described in Section 2.5. The third approach is based on the premise that the basic compatibility problems can only be solved by a completely new start. A proposal in this direction is the Information Automat (Section
2.3 National Software Works

The National Software Works (Figure 2.3.1) is an extension of the ARPANET Project, (Braden 1976). The initial objective of the ARPANET was to provide communication among a number of computer centers so that any person having access to one center could use any of the other computing environments. This helped to simplify the access problem but it did not reduce the other difficulties mentioned above. The user must still know the existence of a tool and the operating system command language of the environment on which it is located before he can start to use it. The National Software Works starts with a premise that the existing tools will remain unchanged and that a "super structure", i.e., a network operating system known as the Works Manager will be placed on top of the present network. This Works Manager will allocate resources, control communications, handle authentication, and maintain a central index of available tools. In addition, each computing environment will contain a front end which is a Command Language Interface to the Works Manager. Each host computing environment which contains tools will have an additional software component known as the Foreman that will act as the interface in using the local tools. In addition, each (tool building) host will contain a file package to provide file compatibility.

This system is currently in development and when completed will make it easier for users of the ARPANET to use tools at other hosts and will somewhat simplify the interface problems among tools by providing one file language. However, it does not help to alleviate the problem of each tool having its own language. In effect, it adds another language and command system which a user must learn though this one will hopefully serve him at many hosts so that he can use local tools without having to know the local operating system language.

The ARPANET is a research vehicle, most of the hosts are research computing centers and most of the users are researchers who are interested in tool development. They are, therefore, less likely to produce Programming System Product Packages and also are more likely to tolerate the limitations of other people's programs when they try to use them. To what extent this experience will be translatable into a production software environment is not clear.

2.4 Software Development Laboratories

A number of "laboratories" have been proposed to aid software development by providing an "integrated" set of tools. These arise in two different situations. The first one is a development group which needs tools and in which management decides some investment in providing an integrated set of tools is desirable. Examples of this are the DAIS system of the U.S. Air Force, (Ruth 1973) (Figure 2.4.1), the SDL Laboratory, Naval Electronics Laboratory Center (1976), and the Support Software System described by Crafford, et al. (1976). An early example is Clear-Caster (Brown 1969). A system which has been in operational use for several years is CADES, produced by ICL for support of production and maintenance of an operating system (Fratten and Snowdon 1976). Davis and Vick (1976) describe the SDS system which is intended for support of development of real time systems. In these situations the relevant tools are identified and some investment is made in making the tools easier to use by providing standard interfaces and providing standard data base systems.

Another class of such systems are ones which are basically designed as programming development laboratories. These include de Jong (1973), TOPD (Henderson and Snowdon 1974), ECL (Wegbreit 1971, Chestam and Wegbreit 1972), and SL-1 (Gardner, et al. 1976).

A proposal for such a system is given by Irvine and Brackett (1976) (Figure 2.4.2). They view their SEF (Software Engineering Facility) as consisting of a System Analyzer, an Interface, a Test and Validate Module, and a Report Generator. All of these interfaces with a Software Engineering Data Base. Other proposals have been made by Hamilton and Zeldin (1976), Walters (1977), and Falls, et al. (1976).

Another example of such a system oriented not so much towards software development, but rather towards providing application users direct access to a number of tools is the GMIS system developed by Donovan (1976). GMIS provides the access through the use of the virtual machine concept and through a generalized relational data base management system.

These software development laboratories form the basis for the Software Support System proposed in Section 3.

2.5 Application Management System (AMS)

The Application Management System (AMS) was developed by TRES, Inc; user experience is described by Wright (1975) and in the EDP Analyzer, Canning (1974). An overview of the system is given in Figure 2.5.1.

The system consists of four major subsystems. Each produces or updates a data base. The first subsystem, the Data Dictionary Generator, accepts data definitions and produces documentation in addition to updating the Data Dictionary. The second subsystem, the AMS Compiler, uses this Data Dictionary and produces a data base of data processing modules from statements in a Data Processing Language. This data base together with the Data Dictionary is used by the third subsystem, the AMS Configurator, to produce an executable system called the Run Control File. The fourth subsystem, the AMS Monitor, executes the Run Control File under the control of the operating system. Each of the subsystems is controlled by a Control Language and each one also produces documentation in addition to updating the AMS data bases.

Wright (1975) describes the use of AMS in the development of two systems LMS and FMS; some information from his paper is summarized in Appendix 2.

---

5 For an interesting discussion of the role of tool development in large projects, see Brooks 1975, p. 127.
2.6 The Information Automat

The Information Automat has been proposed by Wilson (1976). (See also Mills and Wilson (1976).) He begins with a premise that all the support software, including the operating systems, the system utilities, data base management systems, compilers, etc., have been developed independently and each has its own language and its own interfaces. He proposes to develop a kernel system which will be the central software surrounding the hardware to provide these services in an integrated fashion, Hofmagle and Wilson (1976). (Figure 2.6.1). The Information Automat kernel system there will be other layers of support which will provide necessary services for application development and operation. The basic concept is that one language can serve as the communication medium through the automation. A common kernel system therefore will provide the basic support that is now provided by a number of independent systems: TSO, JCL, IMS, GIS, FORTRAN, COBOL, PL/I, OS/VS, etc. The central concept is a language known as a "semantic language" which can be used at each level for each service. While the vocabulary and the actions will vary, the structure will be the same and hence it will be simpler to learn by all those who have to use it.

The fundamental advantage of the approach will be to eliminate duplication. Now each tool in the support systems in today's system must have its own language interpreter and analyzer; in the Information Automat kernel system one will be required. Similarly, all data base management will be done by one program and all the support systems will use the same data base system. This will eliminate the incompatibility among the interfaces of the various support systems. The Information Automat proposal has been described in some detail in a number of documents, but it has not yet been implemented. It is undoubtedly known as a "semantic language" which can be used at each level for each service. While the vocabulary and the actions will vary, the structure will be the same and hence it will be simpler to learn by all those who have to use it.

The approach followed by the National Software Works will undoubtedly make it easier for computer scientists to use tools developed and installed at another computer environment. However, it will probably not be directly usable in systems departments or for software development in which operational software must be produced.

The approach implied by the Information Automat is no doubt ideal and, if implemented, could result both in a substantial decrease of the size of software required for a complete set of tools and a substantial decrease in the cost of using the tools. However, even if it were implemented, there would be problems with incorporating the already existing software produced under previous methods.

One way to solve the interface problem is to have the tools work from a common data base. The approaches to building programming laboratories are usually based on the concept of one level of user using one basic language. To be more useful these systems will have to serve a number of different types of users with a number of different language levels.

What is likely to be practical and applicable in the near future is the development of a user specified facility for software development that makes use of existing operating systems and some other standard software such as data base management systems. Furthermore, this facility should provide not only for the development of new software but also for the maintenance of existing applications software. The specification for such a facility are discussed in the next section.

2.7 Summary and Conclusions

A number of software tools are being developed. However, they are seldom compatible. A user has to learn each of the tools, prepare the input in the form needed, and probably has to use the tools on different computing environments.

Some argue that the software development facilities can now be built, for example, Stillman and Leong-Hong (1975):

"The technology already exists to construct a software production facility. Its components, e.g., advanced programming languages, structured programming precompilers, dynamic analyzers, intelligent text editors and sophisticated file management systems, are within the state of the art. What has not been done is to identify and develop an effective set of tools, integrate them into a single system, and make that system available to a broad spectrum of users."

This, at the present time, is probably a little premature. It is certainly true that the individual components can be constructed. However, the technology to produce an integrated set of tools still remains to be demonstrated. A number of approaches are being tried and more have been proposed as the brief summary given in this paper shows.

The approach implied by the Information Automat is no doubt ideal and, if implemented, could result both in a substantial decrease of the size of software required for a complete set of tools and a substantial decrease in the cost of using the tools. However, even if it were implemented, there would be problems with incorporating the already existing software produced under previous methods.

One way to solve the interface problem is to have the tools work from a common data base. The approaches to building programming laboratories are usually based on the concept of one level of user using one basic language. To be more useful these systems will have to serve a number of different types of users with a number of different language levels.

What is likely to be practical and applicable in the near future is the development of a user specified facility for software development that makes use of existing operating systems and some other standard software such as data base management systems. Furthermore, this facility should provide not only for the development of new software but also for the maintenance of existing applications software. The specification for such a facility are discussed in the next section.

3. Software Support Systems (SSS)

This section outlines the requirements for a Software Support System (SSS) which would provide a coordinated set of tools for software development, operation and maintenance. The facilities discussed are those necessary to develop a new software system to meet a given set of requirements in the environments described in section 1. The specifications outlined in this section are based on the analysis of software development systems (some of which are mentioned in Section 2) and on experience with prototypes of parts of a system, such as the Problem Statement Analyzer (PSA), Telchrow and Hershey (1977).

Then, assuming that a basic Software Support System exists, the next two sections will describe additional methods of decreasing the cost and increasing the quality of the completed products. Section 4 is devoted to methods for reducing the amount of software which must be produced manually by having more of the software produced by the computer. Section 5 is concerned with synthesizing a system out of software which already exists, i.e., out of "reusable modules".

3.1 Software Support System as a Decision Support System
It has become fashionable to characterize certain types of information processing systems as Decision Support System. For example, Sprague and Watson (1977) define them as follows:

"Just recently, information systems with rather unique characteristics have begun to emerge. These systems, usually referred to as Decision Support Systems, feature decision models, a data base and the decision maker as subsystems and are specifically oriented towards supporting decision making."

and state that a

"Decision Support Systems (DSS) should have the following characteristics.

1. The DSS is designed specifically to support decision making. Attention to information flows, report structure, and data base design is specifically related to this primary objective.

2. The DSS is interactive to allow the manager or his representative fast access to models and data. The interactive capability is not necessarily to provide immediate access to minutes-old data, but, rather, to give access to data and models at a speed which matches the thought processes of the manager.

3. The DSS is flexible enough to satisfy the decision making requirements of many types of managers - those in different functional areas, at various managerial levels, and with different management styles.

4. The DSS is an integrated set of data and models which allows the models to work together, and thus avoid suboptimization whenever possible.

5. The DSS is dynamic enough to keep itself up to date without major or frequent ad hoc revisions.

6. The DSS is sophisticated, utilizing modern information processing and management science techniques whenever appropriate."

The Software Support System proposed in this paper can be considered a Decision Support System in the sense that it supports the decision making by System Management (usually a separate department in an organization), Project Leaders, Analysts, System Architects, Software Designers, Programmers, System Operators, Data Base Administrators, Project and Program Librarians, and Target System Users, that is, all personnel associated with information processing system and software system development, use and maintenance.

3.2 Users

One of the most effective ways to ensure that a system will be operationally feasible and economically cost effective is to have one system provide for all the needs of a set of users whose needs are related.

All users should be able to enter the results of their own decisions into the Software Support System data base and then should be able to obtain the information implied by the above characteristics of Decision Support System. An overview of the interaction of the various users with the Software Support System is shown in Figure 3.2.1.

Information Processing Systems management should obtain summary reports describing the status of various projects and various systems. The systems department staff should be able to obtain summary information necessary for the coordination of various projects, evaluation of new projects, and determination of the impacts of completed projects. Project Leaders need to obtain relevant data about projects they manage including both management and technical data.

The Analysts should be able to obtain status and checking reports about the data entered into the data base, including its consistency and completeness. After the analysis is completed, they should also obtain the necessary final reports.

The System Architect should be able to obtain the description of the overall system with which he is concerned and all the information provided by the Analysts. The Software Designer should be able to obtain the requirements produced by the Analysts as well as the specification produced by the System Architect. The Programmer should be able to obtain the up to date specification of the programs he is assigned to develop and the source code of programs of similar functions as well as reports indicating whether all program branches were tested against all requirements and whether the programming documentation is complete. The System Operators should be able to obtain an up to date report on the status and progress of the operated application system (input data preparation, error messages issued, handling of exceptions, action report handling by the target users). During operations, the Operation Staff should be able to record operations data and relate it to the system description. Similarly, the Data Base Designer should be able to obtain the requirements for the data base derived by the Analysts and the System Architect. The Data Base Administrator should be able to enforce data base standards and determine the requirements for the integrated data base. The Project Librarian and Program Librarian together with the maintenance staff, should be able to keep track of all the information regarding the project documentation. It is their responsibility to maintain a complete description of the system and a record of changes that have been made. Target System Users should be able to obtain the description of target systems that pertain to them.

3.3 Data Base

In order to ensure that all the relevant data will be available, it will be necessary to store it in one integrated data base in which the various types of information are separately identified. Thus, the appropriate subparts of the information can be selected for use by the various individuals. As with any data base system, appropriate controls must be exercised to ensure that the integrity of the data base is preserved. Individuals must have access to only that part of the information that they are entitled to and are able to modify, change or augment only the part of the data base which has been authorized for them.
The data base structure will be very complex; for example, Figure 3.3.1 shows the data structure for a prototype of one sub-sub-system of an Software Support System which is designed for the storage, manipulation and retrieval of FORTRAN modules.

3.4 Major Subsystems of a Software Support System

The Software Support System would consist of a number of major subsystems as shown in Figure 3.4.1.

The Requirements Determination System allows Analysts and Target System Users to enter data describing existing and proposed systems. The software performs various checks and analyses and produces hard copy documentation on request. It maintains a data base containing the description of the target system from the "user's point of view".

The Development System allows Architects, Software Designers, Data Base Designers and Programmers to design and construct a system to accomplish a set of requirements recorded in the Requirements Data Base. This system is divided into two subsystems. First, the Software Engineering Facility is designed to permit the rapid and efficient development of a prototype to demonstrate feasibility. This is consistent with the definition of Software Engineering given in Section 1.3.3. Second, the Software Production System is designed to begin with a prototype system and produce a production version.

The Support System allows software systems to be operated and maintained. It is subdivided into two systems. The Operations Facility controls the operation of the target system in an actual environment. It collects statistics that may be used to improve performance. The Installation and Maintenance system provides the capability to install the target system in a particular hardware environment and to maintain it as changes occur. The Management System allows management to maintain plans and obtain data from each of the various project and system data bases. It also provides for all the coordination necessary among projects and systems.

The Data Base System contains all the data that describe the target systems and the projects. It is subdivided into a number of data bases, but the relevant interconnections among all the subdata bases are maintained.

4. Reducing the Manual Effort to Produce Software

4.1 Classification of Methods

In the discussion in the previous section, it was implicitly assumed that the basic tool for software development was an assembly or higher level programming language (FORTRAN, COBOL, PL/1, ALGOL, PASCAL, etc.). The major purpose of the Software Support System was to have the clerical operations performed by the computer. In addition, the Software Support System performed a number of checks and provided management information. In essence the software was still produced manually, though the process was aided by the computer. This section is concerned with methods by which the software can be produced more automatically and, therefore, the manual effort can be reduced. All of these methods basically attempt to reduce the number and effort of manual operations between a statement of requirements and the executable system code. The "decision models" and decision making aids of the Software Support System will be increased and it will become more of a Decision Support System.

The methods for producing executable systems from higher level descriptions may be classified in three ways. One approach to higher level definition is to "specify" larger units of the system. This concept is discussed in section 4.2. Another approach is to present the specifications in a form natural to human beings but which can also be implemented by computer software. These forms are described in section 4.3. The methods by which the executable system is actually produced are outlined in section 4.4. Conclusions on the present state of these methods and trends in their use are given in section 4.5.

4.2 Level of Specifications

Software systems generally may be classified as functional specifications that state what the system is to do but not how it is to do it, at least not in a way that constrains the software structure. There need, therefore, be no correspondence between the structure in the specifications and the structure in the software.

When specifications are given at the subsystem level, each subset of specification will be transformed into a software subsystem. One case where this occurs is where specifications are given in terms of a transaction. The specification for a given type of transaction are implemented as one subsystem within a single type of transaction the compiler may choose its own software structure.

When specifications are given at the module level, each module is implemented as a software module and within a module the compiler decides on the software structure.

4.3 Form of Specification

The form in which requirements are acceptable as input to the computer may be classified as follows:

Natural language. Ideally, any text in a natural language will be acceptable. The software system starts without any knowledge of the problem domain and builds its entire knowledge on what it obtains from the input. Several systems based on natural languages are discussed by Heidorn (1976).

Specialized languages. The structure of the input is still natural language but certain words or phrases have preassigned meanings to the system. These languages are easier to use and the specifications are easier to implement but each language is restricted to a special application field.
Relational Languages. These languages have specific structures and a certain set of reserved words. Input is free format. Examples are the Problem Statement Language (Teichroew and Hershey, 1976) and Requirements Statement Language (Bell, et al. 1977 and Alford 1977).

Forms. Forms have long been advocated as a means of expressing requirements since they are easier to use than languages. Examples of the use of forms are the ADS (Accurately Defined Systems) developed by National Cash Register and TAP (Time Automated Grid) developed by IBM. (A comparison is given by Teichroew 1972). A recent example is the programming language used in The System for Business Automation (SBA) prepared by Zloof and de Jong (1977). An example of a program to produce an invoice is given in Figure 4.3.1. A diagram of the language structure is given in Figure 4.3.2. The language attempts to show the format of the documents graphically. The domain over which the program is to be executed is shown by underlining an item in a table (which is considered to be a relation).

Decision Tables. Decision tables have been advocated as useful tools for specifying requirements. However, there appears to be general agreement that they are not widely used. See, for example, Chvalovsky (1975) and Central Computer Agency (1973). One of the reasons given in the latter report is that decision tables by themselves are not enough, they should be incorporated into a more comprehensive system. Attempts to do so have been made in HSL/1 developed by Hoskyns (Rhodes, 1972) and marketed by Martin Marietta and in AMS developed by TRS5, Inc. (Wright 1975). The HSL/1 approach to programming is contrasted with the traditional method in Figure 4.3.3. (The AMS system is briefly described in section 3.3.)

Questionnaires. The questionnaire approach is simply a list of multiple choice questions. A specification consists of the selection of one of each of the available choices. The first formal description of the technique appears to the one given by Ginsberg, et al. 1965. The combination of this technique with decision tables was advocated by Low (1973):

"Programming by questionnaire combines aspects of decision table programming and general purpose programming by using decision tables to construct an application program through the selection of certain source statements from a predefined file. It is proposed that programming by questionnaire is a useful compromise between general and special purpose programming for a significant class of large scale problems."

The most extensive use of the technique appears to be by IBM in the Application Customizer which is used to produce executable code for System/3.

Graphical Languages. Graphical languages appear attractive as a means of specifying requirements since human beings can comprehend a large amount of information when it is presented graphically. Graphical languages have been proposed as supplementary forms to other languages, (see, for example, Young and Kent, 1961). When recorded manually, however, graphical languages must be transformed into input suitable for computer processing. Few organizations yet have extensive capability to input graphical information directly and, therefore, graphical input is not widely used. This conclusion is also reached by Rose (1976) in relation to specification of hardware:

"Computer hardware description languages (CHDL's) will continue to develop and to be used as "front ends" of conventional hardware design automation systems, but the research emphasis will have been on system descriptive languages such as that used by LOGOS. It is probable that these languages will be string-oriented rather than graphic because of the cost of sophisticated terminals."

Module Specification Languages. These languages focus on specifications at the functional level and usually include some facility to describe relationships among modules. Examples are the languages proposed by Bin (1976), Beck (1976), and DeRemer and Kron (1976).

Augmented (Procedural) Programming Languages. A number of extension to procedural language have been proposed. Some of these incorporate structured programming constructs, for example RAPTOR (Kernighan and Plauger, 1976). Others are designed, not so much to reduce the amount of manual programming, as to provide facilities to state requirements which will make the object code more reliable. Another augmentation is in the direction of making the language extensible for data types (Liskov 1975).

4.4 Production of Software Systems

The methods discussed in this section are designed to implement higher level specifications (at one of the levels outlined in 4.2.), in one of the forms described in 4.3, and produce an "executable" system automatically with minimum human intervention. For the purpose of this section, it does not matter whether the "executable" system is object code for a given computing environment or whether it is source code in a programming language. In practice, of course, the performance of the system may very well be affected by what form of executable system is produced.

4.4.1 Artificial Intelligence. The input to this method is either text in a natural language or in a specialized natural language. A survey of the state-of-the-art in automatic programming natural language dialogue is given by Heidorn (1976). He begins with this introduction:

"Since the early days of computing, effort has been put into automating more and more of the programming process. The ultimate objective in automatic programming is to produce a system that can carry on a natural language dialogue with a user (especially a nonprogrammer) about his requirements and then produce an appropriate program for him. Although the basic idea of 'programming in English' has often been expressed in the literature, only in recent years have any serious attempts been made toward producing such a system."

He then describes four experimental systems and discusses a number of research issues and
concludes with the statement:

"Higher level considerations such as this will have to be dealt with in addition to the more technical issues discussed above before natural language automatic programming can become a practical reality."

One of the systems surveyed is the SAFE system being developed by Belzer (ISI Research Staff 1976). He describes his approach as follows:

"Only modest gains in programming productivity have been produced in 25 years of software research, but the groundwork has been laid for major advances through rationalization and automated aids. This groundwork rests on two critical ideas: that specification must be separated from implementation, and that the separation between these two processes should be a formal operational abstract (i.e., very high level) program rather than a non-operational requirements specification. Structured programming represents the first results of combining these ideas. It is a special case of a more general two-phase process, called Abstract Programming, in which an informal and imprecise specification is transformed into a formal abstract operational program, which is then transformed into a concrete (i.e., detailed low-level) program by optimization. Abstract programming thus consists of a specification phase and an implementation (optimization) phase which share a formal abstract operational program as their common interface."

He summarizes the current status as follows:

"Though the results using this example are promising, and although we have attempted to build a general system capable of handling a wide variety of specifications from many different domains, it is extremely difficult to extrapolate from a single data point. We therefore are planning to present several different and more complex examples to the system during the next year."

There seems to be general agreement that while this approach is promising and research should be continued, it will be some time before this method is capable of producing results that compare with what now is being produced manually.

4.4.2 Software System Generators. The concept of generating complete systems from a set of requirements has long been a dream of application software developers. Recently, the generator concept has been described by Dolotta, et al. (1976). Their description is given in Appendix 3.

The system generator approach differs from the artificial intelligence approach primarily in that some knowledge of either the application area or the software system structure is assumed. The domain is therefore much more limited and specialized.

Three types of software system generators may be considered. In the first, the generator is capable of producing modules and assembling them to accomplish the stated requirements. In its full generality, this is equivalent to the artificial intelligence approach. In practice, however, the domain must usually be constrained in some way.

A second approach assumes the existence of a number of modules. The generator need only to decide which modules are necessary and provide for the required interconnections. The system based on "Programming by questionnaire" (Ginsburg (1965), Low (1972)) is an example of this level of generators. Another example is automatic program synthesis (Feldman (1972), and Lee, Charnig and Waldinger (1974)).

A third level occurs when a skeleton of the system exists and the generator merely parameterizes it to fit a particular environment. The system generation of an operating system for a particular environment is an example of this level.

4.4.3 Module Preprocessors. The module preprocessors work at the module level, i.e., each module of a specification is transformed into a software module. The methods differ by the degree to which the specification is rearranged or transformed by the preprocessor. In the system proposed by Beck (1976) for example, the specification given in a module specification language are transformed into "Standard Processes." In general, these methods are based on producing a directed graph from the specifications and then partitioning the graph in some way.

4.4.4 Compilers for Augmented Programming Languages. These software systems accept statements in augmented languages (including structural programming constructs and abstract data types) and produce either source statements in the programming language or object code. These compilers expand the augmented languages but generally do not rearrange statements to the extent done by the Preprocessors.

4.5 Conclusions

The preceding discussion indicates that a number of approaches for producing executable systems from higher level specifications have been proposed and/or are being developed. These use different levels of specification, different forms of specification and different approaches to the generation or executable systems. However, the use of these techniques, to reduce the amount of software that must be produced, is so far limited.

System Generators are the method with the best chance to yield a dramatic reduction in the cost of the software components of systems and at the same time improving the effectiveness of the system for the user. However, before System Generators achieve widespread use, a number of problems will have to be overcome. The most pressing problem is introducing methods for persuading the organizational unit that needs a new system, to accept one that is generated rather than custom built. One of the necessary conditions for acceptance of the generated target system is that it, in fact, accomplish the requirements. Thus, considerable skill is required to produce generators that meet a wide range of user's to justify the investment.

Under the right circumstances, these approaches may contribute substantially to reducing the cost of high quality software. Thus, it is worthwhile
5. Reducing the Amount of Software that must be Produced

5.1 Introduction

There are a number of methods, techniques, and approaches that will reduce the amount of "new" software necessary to meet a specific requirement. A person or organization who has a requirement can obtain software to meet his needs by one of the ways identified in section 1.2: develop the software himself, have it developed for him, select from a set of available packages, or select it from a set of available "turn-key" systems.

The greater the proportion of the software that is already available, and therefore, does not have to be developed, the more the user gains. The software should be less expensive since the development cost is spread over more users, it should be available sooner since it does not have to be developed, and it should be more reliable since it has presumably already been tested.

The developer of software should also gain from using predeveloped software; he should be able to improve his profit through increased sales since each user can obtain the software at a low cost, he would be able to increase the amount of development he does since he becomes more competitive as compared to the custom-tailored developers, and he can afford to invest in improving his development methods along the lines described in sections 3 and 4.

The potential advantages of reusable modules is illustrated by the price charged by one software company, Martin-Marietta, as shown in Appendix 4. According to its advertisements, the company charges $6 per statement for custom-tailored software but only $1 per statement if the system is built entirely out of existing modules.

Despite the apparent advantages of making maximum use of existing software and of designing and building software to have a maximum number of users, a large amount of the development today is concerned with designing and building new custom-tailored systems to meet each requirement. This section examines the present status of "generalized" software and considers some of the problems that must be solved if the concept is to be incorporated into the Software Support System.

Section 5.2 discusses attempts to use existing software where the software itself can be obtained at nominal cost. Section 5.3 outlines very briefly the current status of the software industry which, by definition, is motivated to produce software with as broad a market as possible. Section 5.4 describes one attempt to develop a method to synthesize systems from modules based on application functions. Section 5.5 and 5.6 outline attempts, at national levels, to develop systems based on reusable modules in the USEE and Japan respectively. Section 5.7 and 5.8 describes some of the issues that must be addressed and outlines some possible approaches in reusing modules and generalized software respectively.

Section 5.9 summarizes how software development in a Software Support System, as defined in Section 3 and augmented by the software generator techniques discussed in Section 4, can be further improved by incorporating the concepts of reusable modules and generalized software.

5.2 Software Distribution

A great deal of software is available for use at a nominal or even zero cost. Nevertheless this software is not widely used. The reasons for this include all the reasons given in section 2 for the specific case of software for development of application software. Large organizations frequently have programs to disseminate knowledge of available software. For example, ComputerWorld recently reported on one attempt by the U.S. Government to foster sharing of software (Appendix 5).

5.3 Generalized Software Produced by the Software Industry

The software industry markets program packages which are usually referred to as generalized software since a package is intended to satisfy a large number of users with as little change as possible. Many organizations which develop software for their own use also attempt to sell it to others as generalized software. While many packages are available very few of them achieve any substantial number of installations. The most successful is Mark IV which is marketed by Information, Inc. According to Datamation (December 1976, p. 142) there are over 1000 installations of Mark IV which have resulted in more than 40,000,000 dollars in revenue. Analysis of other packages that have been purchased by many users, shows that the most success has been achieved in application-independent packages.

Data base management is an example of an area where few organizations can now justify producing a software system for themselves. However, software enterprises have had limited success in marketing generalized application packages.

5.4 Modularization Based on Decomposition of Application Functions

The synthesis approach to software system development attempts to realize a system to accomplish a given set of requirements from a given library of available components. There have been a number of attempts to use this approach though few have been reported in the literature in any degree of detail. One attempt was the Information Systems Engineering effort described by Welsh (1968). It is an instructive example because it involved a relatively large effort sustained over a number of years, because it was concerned with application modules, and because it did not succeed in replacing the conventional custom-tailoring approach.

The effort was motivated by the attempt of a large industrial firm to avoid building unique systems to accomplish essentially similar requirements (payroll, production scheduling, inventory control, etc.) in each semi-autonomous subsidiary. (The first approach was to try to build a system to accomplish a given requirement at one subsidiary and then transport it to the others - this was abandoned as impractical after a few attempts). The approach, summarized in Figure 5.4.1, uses three tools:

- Functions - predefined units of information
The synthesis consists of three major steps:

a. The requirements for a system are determined and expressed using the Configurator.

b. A solution to the needs is synthesized using the Functions and the available Satisfiers. If no acceptable Satisfier is available for a given Function, a new one is produced.

c. The system is "assembled" and is then ready for installation and test.

A large part of this development effort consisted of identifying and specifying the Functions. The criteria used to identify the Functions were:

- the Functions had to be mutually exclusive,
- the Functions had to be exhaustive,
- the Functions had to be of approximately the right size.

Approximately fifty functions were identified and described. In some cases a description required several hundred pages. The functions are listed in Appendix 6.

Despite the extensive development effort and intensive attempts to apply the results, the approach is not being used. The reasons are partly technological. Functions that covered data processing, e.g., Message Discrimination and Validation, File Update and Surveillance and Report Data Compiler, were apparently implemented in such a way that the synthesized system could not compete in usage of computer resources with custom tailored systems.

However, the main reasons for the lack of adoption were institutional. The managers of the subsidiaries could not be convinced that there would be sufficient gains to warrant the loss of autonomy that they had with the custom tailored systems. Corporate management on the other hand was not convinced that corporate benefits would be sufficient to warrant forcing the subsidiaries to use the synthesis approach against their will.

5.5 System Development in the USSR

The USSR is engaged in a major effort to develop a relatively large number of computerized management information systems (Myasnikov 1974, pp. 88,95). In order to facilitate the development procedure, detailed guidelines have been established and approved by the Union Committee for Science and Technology of the USSR Council of Ministers (1972). The development procedure is based on the concept of synthesizing systems from a number of existing modules rather than custom building a system for each application. The need for standardization of the modules is widely emphasized (Machrov et al. (1974), p. 281 and Myasnikov (1974) p. 92). Besides the long-existing Central Institute of Scientific Research in Management and Control Technology in Minsk (employing several thousand researchers), in 1974 the computer industry created in Kalinin a Software Development Center charged with developing, from standardized modules, application packages serving all the potential users in the USSR, (Myasnikov (1974) p. 92). The users are strongly urged to build "open" application systems, which could be enhanced in the future when the basic system has already been in operation for some time. The wide use of the first 130 standardized application packages has been recommended by the Union Committee for Science and Technology of the USSR Council of Ministers (Myasnikov (1974) p. 92). The cooperation in this field between the USA and the USSR has become feasible since the same Committee recommended COBOL, ALGOL, FORTRAN and PL/1 as basic programming languages and the IBM standards have been adapted for the drop-in replacement of the RIAD Computer Series (Myasnikov (1974) p. 91).

5.6 System Development in Japan

In 1973, the Japanese Ministry of International Trade and Industry (MITI) funded a project to make it easier, faster, and cheaper to produce application programs from small chunks of previously written and stored application modules, and assemble these into working, or nearly complete, new programs. The results as reported by Datamation, (September 1976 p. 97), were as follows:

"The five groups, however, each did their own thing with little, if any, management or coordination from a central body. There was not even agreement on the definition of a module. At least one group wrote its modules in FORTRAN. Another used DPL. And one group really didn't know what the others were doing. The result, say some observers in Japan, was a flop."

MITI in 1976 created a new corporation, Joint System Development Corporation (JSD), which has initiated a five year program to produce a Program Productivity Development System (FPDS). An overview of the initial system is given in Figure 5.6.1 (JSD, 1976).

5.7 Issues in Reusable Software

In dealing with a complex system, either in analyzing an existing system or in building a new one, it is often beneficial to treat it as a hierarchy of modules comprising the entire system (Simon 1962). The use of subassemblies in manufacturing industry is a well-known engineering application of this principle. With regard to information processing systems, modularization has long been recognized as desirable, and many papers have been written on modular programs, modular software, modular file organization, and modular systems architecture. However, many systems when completed are not, in fact, modular. There are, therefore, many factors which tend to work against effective modularization even if the designers have that as their goal. Some argue that modularization will be achieved only if it is forced and there is no other choice.
5.7.1 How to Modularize. Any given system can be decomposed into smaller systems or components in many different ways. A number of methods for decomposition have been proposed. Some of the problems are:

1. Whether to modularize on application-independent functions or on application dependent functions. Frequently an examination of application software will reveal some application independent functions that are repeated several times. The identification of these functions and the provision of modules to accomplish them when they are required usually leads to a reduction in the amount of application software that is needed. See for example IBM(1977).

2. How to separate data definition from processing definition. In the early days the lack of distinction between data and instructions was regarded by many as one of the great achievements of the electronic computer. It has become increasingly clear, however, that the distinction should be made and the trend is clearly towards further and further separation.
   - COBOL separated the Data Division from the Procedure Division, thus allowing each to be defined, and hopefully modified, separately though they have to be compiled together.
   - Data Base Management Systems carry this concept one step further by having separate Data Definition Languages and Data Manipulation Languages.
   - Data Dictionaries provide for definition of data elements that contain the user oriented descriptions as well as computer oriented descriptions.
   - Another trend is to distinguish data that describes the structure of the organization, and is used to control processing, from other data. In its past the computer has frequently been embedded in the programs; the objective now is to remove them from the program and put them in tables so that the tables can be changed without changing the program.

3. How to separate processing from control-flow. By separating modules which perform basic functions from the logic which causes one function to be performed rather than another, the likelihood that the basic modules will be reusable is increased.

5.7.2 What is the Best Size for a Reusable Module. One of the basic questions in reusable module development is the size of the component of software which should be standardized or made reusable. The terminology, referring to size of software components, used here is defined in section 1.1.2. All of the levels system, including subsystem, program, routine and module have been used as all of reusable components.

Many programming languages have provisions for incorporating predefined sets of statements, usually called macros. One of the very first concepts in software development was that of the closed subroutine. Almost every installation has, over time, developed a library of subroutines which perform common useful functions, and most programming languages have adopted standard set of subroutines such as SIN, COS, etc. The extension of this principle to the program level, results in the many utility functions which are generally available often at the system command level for copying, sorting, editing, etc. Taking advantage of these well-defined and tested capabilities not only simplifies the task of the system developer, but makes the developed system easier to understand and maintain.

Turn-key systems are acquired in cases where the buyer does not have the capability to produce the system himself or where he is prevented from doing so by legal or policy constraints. The former occurs primarily in small organizations which up to the present, have absorbed only a small part of the computer industry's products. This will probably be relatively unimportant for the software industry since the turn-key systems are likely to make maximum use of firmware. The latter occurs in some government and business installations and in some industries under government regulatory control. These situations are also likely to remain relatively unimportant. The second situation arises when the cost of a turn-key system is so clearly less than that of a custom built system that the buyer has no choice. This requires that the buyer know the relative costs of both turn-key and custom made systems including maintenance and other related costs.

While software components at all these levels can be made reusable, the level with the most potential for mass usage is the "module" level. The term "module", as defined in 1.1.2, is a component of software that can be compiled separately but may contain components which can also be compiled separately.

5.7.3 Efficiency. It is generally accepted that modular systems are inefficient because of the overhead involved in linking and calling modules. Camp and Jensen (1976) have investigated the system cost or overhead associated with software modularity. They conclude that the added cost is relatively small compared to the savings gained in software development and maintainability. The overhead that currently exists can probably be reduced in future systems by more effective linking and loading.

5.7.4 Parameterization. All generalized approaches, even turn-key systems and those based on modules, require adaptation to an individual installation for three major reasons: hardware configurations are different and change over time, the individual installation requirements differ, not all want all options, and the software itself is changed to correct errors and improve performance. It is therefore desirable to embed as few parameters in a module as possible.

The early information processing systems were stand-alone applications with their own files. This resulted in considerable duplication of data, duplication of processing operations, and great difficulty in maintaining the systems. For example, one major food processing firm had to rewrite its entire order entry, production control and distribution software when a new product was introduced because the knowledge of the existing
product codes was embedded in the individual modules. Flauger (1976) describes the difficulties caused by lack of parameterization in the New York Times project.

5.8 Issues in Generalized Software

The individual programmer should organize his programs so that he can reuse portions of his code without having to rewrite it. However, the components are usually not used by others since they are not documented well enough. If components are going to be reused they must be developed with this in mind. Provisions must be made for component definition and development, description, utilization of modules is specification of what each module should do. An approach which allows each software designer to develop modules and add them to the library will not be successful. Three requirements are necessary. First, some set of criteria will have to be established for the specification of a module. Second, the modules will have to be adequately described, catalogued, and documented so that a prospective user will be able to locate those that might satisfy his needs. Third, the modules will have to be constructed according to some standards for both internal structure and external interfaces.

Generalized systems have had only limited success. If these methods are to be pursued, the reasons for the lack of success must be identified to determine whether they can be overcome or whether the evolution in other areas will change conditions so as to favor generalized systems.

The concepts of reusable modules and generalized software can be incorporated in the Software Support System. This may not immediately and/or completely overcome all the difficulties identified in section 5.7 and 5.8, but considerable improvement can be made. The use of a data base to record all system descriptions automatically enforces a standard definition of a module. Furthermore, the data base contains descriptive information about each module. This will simplify determining whether a module that accommodates a desired function already exists.

Each module will be stored in a data base together with its external description. When a new module is entered into a data base it will be examined and its external description verified as being consistent with its internal operation. It would be possible also at this time to search for possible overlap with existing modules.

6. Summary and Consideration for Progress

6.1 Summary

The concern of this paper has been with surveying the state-of-the-art in software, examining the attempts that have been and are being made to improve software, analyzing the reasons why these attempts have not always been successful, and finally, synthesizing an approach which may overcome many of the lack of success in the past. The objective is to propose methods to obtain the same degree of improvement in software as that which has been obtained in hardware. The hardware improvements in capability, reliability and reduction in cost have been achieved through research, separation of engineering and manufacturing functions and mass production. These same concepts can be applied to software, though currently not as readily.

The environment in which these techniques are most relevant is one in which a large amount of software must be produced and maintained over a long period of time during which the requirements and computing environment may change. A software system in such an environment is usually large with hundreds of modules and tens of thousands of lines of source statement. The software must be developed and maintained by staff whose membership is also changing. The software may exist in many versions each with many releases. The techniques are not necessarily appropriate for situations in which state-of-the-art software is produced by experts primarily for research purposes. In these situations software is not intended to be used for operational uses and does not have to be maintained. Cost and performance may not be as important as demonstrating feasibility.

6.1.1 Major Components. Software can be provided to the user at a substantially lower cost, be more reliable and more adaptable as his environment changes, by an approach that consists of three components.

Coordinated software development tools to provide assistance to the professional software developer. The assistance will be clerical, for enforcing conventions, and managerial. The clerical function will provide storage of the software components and preparation of hard copy documentation. The system will ensure that conventions are being followed, will provide for performing a check of various kinds on the software components. It will also provide managerial functions for keeping track of progress and for evaluating the impact of changes. The system, at this point, however, does not produce software; the software is still produced manually but through the use of the computer as an automated information storage and retrieval system. More detailed specifications are given in section 3.

Increased use of the computer to reduce the amount of software which must be developed “manually”. These techniques permit the computer to be used to reduce the manual work of actually generating the software and also support decision-making by all those involved with software. The techniques are described in section 4.
Increased use of existing software to reduce the amount of new software that must be developed in a given system. One of the most effective ways to reduce cost and time is to make use of software that already exists and has been fully debugged and tested. The techniques for accomplishing this are discussed in section 5.

In the past, individual approaches in most of these areas have been tried, but progress has been slow. The impediments to progress are not limited to technical problems, though these do exist. More serious are the management, economic, and political problems. Analysis of the various techniques indicates that many of these limitations can be overcome by an integrated approach rather than considering them to be mutually exclusive.

6.1.2 Implementation Objectives. The creation of comprehensive Software Support System should be based on the following concepts:

- An integrated, coordinated data base that contains all the relevant information about the software. The more complete the relevant information the more the data base will be able to serve all the needs of various users of the system. A coordinated data base also eliminates the difficulties caused by incompatible interfaces.

- The system should serve as much of the information needs of all the individuals concerned with systems. The more these needs are met, the more likely it is that the individuals will use the system and provide the necessary input data.

- The system should provide for continual evolution. It should be able to accommodate the software which already exists and which must be maintained. It should also be able to provide for future transition to other environments. This includes both hardware and hard software such as operating systems and data base management systems.

- The system should have a consistent approach to languages. This will make the entire system easier to understand and reduce the time required to learn to use the system. It will also reduce the cost of implementing the system and extending the functions it provides.

- The system should provide consistent interfaces among the various functions that it provides. The reports should have standards formats so that the contents of one report may be used as the input to another.

6.1.3 Potential Benefits.

A Software Support System with the capabilities outlined above has the potential to provide the same improvements for software as has been obtained in hardware. These will become particularly evident over time as a data base of reusable modules is built up and as the techniques to generate software from a higher level specifications are gradually refined.

In time, the quality of the software produced should be so superior that production software produced by these methods will replace software developed by traditional methods. This is exactly the way that commercially produced data base management systems have completely replaced individually developed data base management systems.

6.2 Considerations for Progress

It is obvious that a development of a Software Support System with the capabilities described above is no small undertaking. The technical, operational and economic feasibility of such a system must be examined and the specific actions that are needed for such a system to become viable determined.

6.2.1 Technical Feasibility. The first requirement is the ability to maintain in a data base, all information about the software system to be developed. Data base management systems are in existence and have been used in cases such as PSA (Teichrow and Hershey, 1977) and REVS (Davis and Vick, 1976). The amount of information that has to be stored is less than that of many applications in which these data base management systems are now being used. The performance requirements of such systems need to be determined, though again this is not expected to be any major difficulty. It must be recognized, however, that a Data Base Administrator function will be necessary in this application as it is in other applications using data base management systems.

One of the major requirements is to develop an integrated approach to the use of languages for specifying the input to the system, and for commanding it to perform the necessary operations. A consistent approach is necessary as there will be many different users and each requires a language that includes his own vocabulary. A large number of languages, however, become difficult to learn and difficult to maintain. A number of different base languages are in existence and have been discussed in section 3. It appears that natural languages will not be suitable since natural language processing is still too difficult. The more promising approach is the use of a relational language such as that used in PSL and RSL, supplemented by one or more levels of procedural programming languages. This approach has the advantage that the terms in the language can be selected so as to be natural to the user. The approach of the specifying objects and their relations is easy to follow and applies to most of the areas which the system is to be used. It will of course be necessary for the data base administrator to insure consistency in the naming of objects and relationships. Thus, in the interrelationships among objects specified by different users can be maintained. The basic relational language can be supplemented by forms, tables, graphical language, etc. as desirable.

It will be desirable to evaluate these languages further to improve their ease of understanding, power of communication, and human engineering.

One of the major advantages of the Software Support System will be the ability to generate executable systems for different computing environments. The techniques available for this have been discussed in section 4. Many of these techniques require further evaluation; however, with the basic approach of the data base and the languages mentioned above, these can be incorporated as they are being developed and proved. The desirability of developing systems that operate in different computing environments can be improved by the techniques which were
identified in section 5.

A number of current limitations in progress in reusing software are identified in section 5. The Software Support System requires, and implies, that all software components can be named. This is the first step in being able to identify and reuse software. The system encourages descriptions that will be helpful in reusing the software. Nevertheless, additional discipline will have to be provided by management to ensure that the adequate description exists for the identification of the potentially reusable software components.

6.2.2 Operational Feasibility. Even if a Software Support System is built there remains the question of whether it would in fact be used. Clearly, such a system is first and foremost a management tool. Its basic objective is to reduce the cost and improve the quality of software from the point of view of the organization that is producing it. Therefore, it is absolutely essential that the approach has full management understanding and support.

However, the major users of the systems will be the professionals: Systems Analysts, System Architects, Software Designers, Data Base Designers, Programmers, Quality Assurers, etc. These individuals will use it if it provides functions that they recognize as important, which management recognizes as important, and which the system performs more effectively than if they were done manually. This implies that in the design of the system considerable attention must be given to each one of the functions if the system is to be cost effective.

The use of the system will also depend to a good extent on its ease of use and hence, considerable attention must be paid to making the system as simple to use and as reliable as possible.

Further research and development is required into what functions should be accomplished and how they can best be performed in the system. On the basis of experience gained with the prototypes, such as PSA (which at present provides only a part of the functions of a full Software Support System) it is evident that it is possible to design a system that such professionals will use.

6.2.3 Economic Feasibility. The System clearly requires substantial investment and the initial problem is to obtain the necessary resources. One approach is to build the system software in phases. Thus, benefits are obtained from the beginning and since the system is open ended, compatible components can be added as they become available.

Even if it is available, can such a system compete with present manual methods? There are clearly costs associated with the use of such a system which includes training, software, maintenance, documentation, etc. These costs are all highly visible. The benefits, on the other hand, in a particular application in a particular installation, are likely not to be as visible. They will be hard to measure, and take effect only over a period of time. However, it can be argued that, in general, the out-of-pocket direct costs of producing systems with such a facility should be no more than the present direct costs if they were completely measured.

6.3 Conclusion

For an organization that is continuously developing and maintaining software, the use of a Software Support System should result in substantially lower costs and better overall quality. As this use becomes completely dominant, the benefit of reusing systems and being able to generate systems with less and less manual intervention will become more evident. If these design objectives are successful, the system should eventually become indispensable.

References


Blosser, Patrick Alan, An Automatic System for Application Software Generation and Portability, Ph.D. Dissertation, August 1976, Purdue University.


Canning, Richard C., "Are We Doing Things Right?", *EDP Analyzer*, Vol. 13 No. 5, June 1975,

Canning, Richard C., "Are We Doing the Right Thing?", *EDP Analyzer*, Vol. 13 No. 5, May 1975,

Canning, Richard C., "Do We Have the Right Resources?", *EDP Analyzer*, Vol. 13 No. 7, July 1975,


*Computer Weekly*, "Hewlett-Packard enlists the marketing aid of Hoskinsa," July 15, 1975


Datapro, *Directory of Software*, Datapro Research Corporation, New Jersey


Ginsberg, A.S., Markowitz, H.M. and Oldfather, P.M., "Programming by Questionnaire," RM-4460-RR, RAND CORP., Santa Monica, Calif. 1965


- 209 -


Hofnagle, G., M., Wilson, "The Kernel System of the Information Automat," IBM, 1800 Frederick Pike, Gaithersburg, MD 20760, April 1976


ICP, ICP Quarterly, International Computer Programs, Inc. Indianapolis, Indiana


Leavitt, Don, "Center to Aid Sharing of Federal Software," Computer World, 1976


Lucca, Carlos J., Cowan, Donald D., "Toward a System's Environment for Computer Assisted Programming," Department of Applied Analysis & Computer Science, University of Waterloo, Waterloo, Ontario, Canada, CS-76-06


Myasnikov, V. A. "Computer-aided management today - Experiences, problems and prospects." Article written by the Head of the Board of Computer Technology and Management and Control Systems at the Union Committee for Science and Technology of the USSR Council of Ministers, EKO - Economika and

Naval Electronics Laboratory Center, "Systems Design Laboratory Preliminary Design Report," San Diego, California, NELC N725 TN345


Reifer, D.J., "Interim Report on the Aids Inventory Project," The Aerospace Corp., El Segundo, California, SAMSO-TR-75-184, 1975b


Ruth, John C., "DAIS: The First Step," NACCON '73 Record


Takasaki, I., EDP-IT's Production and Application in Japan, Proceedings Third Annual Computer Applications Symposium, Chulalongkorn University, Bangkok, Thailand, July 1972.


Union Committee for Science and Technology of the USSR Council of Ministers - General methodological guidelines for development of computer-aided management systems in enterprises. ( Gosudarstvienyj Komitet Sovieta Ministrov SSSR po Nauke i Tekhnike - Obshchestvennuyye materialy po sozdaniu avtomatizirovannykh sistem upravleniya predpriiatiami (ASUP), Minsk 1972.)


Wilson, Max, "The Information Automat," Proceedings, SHARK XLVII, Session L103, Montreal, August 15-20, 1976


Wright, A.W., "User Experience with the Application Management System (AMS)" Info '75, New York September 9, 1975.


Appendix 1

Types of Tools Defined by Reifer (1975a)

1. Accuracy Study Processor
2. Analytical Modeling
3. Analyzer
4. Automated Test Generator
5. Automated Verification Systems
6. Bootstrap Loader
7. Comparator
8. Compiler
9. Compiler Building System
10. Compiler Validation System
11. Compressor
12. Consistency Checker
13. Constants Auto Checker
14. Correctness Proofs
15. Cross-Assembler
16. Cross-Reference Program
17. Data Base Analyzer
18. Data Definition Language
19. Data-Exception Handler
20. Decompiler
21. Design Language Processor
22. Diagnostics/Debug Aids
23. Driver
24. Dynamic Simulator
25. Editor
26. Engineering/Scientific Simulations
27. Environment Simulator
28. Emulation
29. EQUATE Program
30. Extensible Language Processor
31. Flowchart
32. Generator
33. Hardware Monitors
34. Instruction Simulator
35. Instruction Trace
36. Interface Checker
37. Interpreter
38. Interrupt Analyzer
39. Language Processors
40. Library
41. Linkage Editor
42. Linking Loader
43. Loader
44. Logic/Equation Generator
45. Macroprocessor
46. MAP Program
47. Modular Programming
48. Overlay Program
49. Peripheral Simulator
50. Postprocessor
51. Preprocessor
52. Process Construction
53. Production Libraries
54. Program Flow Analyzer
55. Program Sequencer
56. Report Generator
57. Requirements Language Processor
58. Requirements Trace
59. Restructuring Program
60. Scoring Program
61. Simulator
62. SNAP Generator
63. Software Monitor
64. Standards
65. - 212 -
Appendix 2

"APPLICATION PROGRAM DIMENSIONS"

<table>
<thead>
<tr>
<th></th>
<th>LMS</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM/MODULES</td>
<td>527</td>
<td>474</td>
<td>1,001</td>
</tr>
<tr>
<td>SOURCE STATEMENTS EXECUTABLE</td>
<td>166,488</td>
<td>86,851</td>
<td>253,139</td>
</tr>
<tr>
<td>COMMENT/DOCUMENTATION</td>
<td>20,538</td>
<td>28,221</td>
<td>48,759</td>
</tr>
<tr>
<td>TOTAL</td>
<td>187,026</td>
<td>114,872</td>
<td>301,898</td>
</tr>
</tbody>
</table>

"PRODUCTIVITY ANALYSIS"

<table>
<thead>
<tr>
<th></th>
<th>LMS</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELAPSED TIME</td>
<td>16 Mos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MANDAYS EXPENDED</td>
<td>1,235</td>
<td>2,375</td>
<td>3,610</td>
</tr>
<tr>
<td>TOTAL SOURCE STATEMENTS</td>
<td>166,488</td>
<td>86,851</td>
<td>253,139</td>
</tr>
<tr>
<td>AVERAGE SOURCE STATEMENT PER MANDAY</td>
<td>151.4</td>
<td>48.4</td>
<td>83.6</td>
</tr>
</tbody>
</table>

NOTE: MANDAYS ARE ADJUSTED FOR 40-HOUR WEEK AND 21 DAYS PER MANMONTH

"DEVELOPMENT BENEFITS"

- simplicity
- reduced amount of code AMS language facilities eliminates application programming
- standard system architecture eliminates part of design phase
- documentation machine generated as development progresses
- problem resolutions faster - debugging and testing aids

OPERATIONAL BENEFITS

reliability
maintainability
extendability
non-obsolescence

Appendix 3
Application Generator Proposed by Dolotta (1975)

"The primary purpose of this study is to point out those requirements of the end users that imply significant research or development efforts, rather than to attempt to present complete solutions to meet such requirements. It seems appropriate, however, to outline an example of the type of solution that could meet the rather severe set of requirements given above.

One can envision an "applications generator." This tool can be implemented in several different forms, each meant for a specific type of application (e.g., payroll), or for a class of applications. The developer and the applications generator would interact via a "menu" of alternative selections (primarily multiple choice, or true-false) on a television-like console. The generator would view each application as a hierarchically organized set of building blocks that are to be tied together in a manner dictated by the choices offered by the generator and selected by the developer.

At the beginning of the interaction, the choices offered to the developer would consist of high-level function or general intent alternatives. Further on in the development "session," the offered alternatives would be at more detailed functional levels, until input formats, output formats, data base requirements, etc., would be specified at the most detailed level. At every level, however, alternatives offered to the developer by the generator would be limited to those consistent with the higher-level selections already made earlier in the "session."

Such a generator cannot, of course, contain all conceivable functions that will (or might) be desired. This introduces the requirement for always being able to select an alternative that is "none of the above." In this case, the developer would write a new module (or a hierarchy of new modules) to fulfill the new requirements. This would be done in a highly standardized and structured manner, so that, once implemented and tested, the new functions would be automatically added to the applications generator and could then be selected by subsequent developers.

- REDUCED SKILLS LEVEL
  - simplicity
  - AMS verbs, access methods to perform complex DP functions
  - standard "building block" architecture to guide personnel
  - tabular decision logic eliminates complex implicit logic
  - high level, automated, logical debugging and testing aids

- INCREASED PRODUCTIVITY
At the completion of the specification process, the generator would bind together the modules of code (which could consist of a combination of object modules, generated source modules, and interpretive modules) necessary to perform the specified functions. This could (and almost always would) be followed by an interactive checkout phase, where either automatically generated or user-specified test data would be run through the resulting applications system for verification by the developer and by the end user.

It seems quite feasible to develop such an applications generator for a specific type of application, such as inventory control. Although this could have a significant impact on an industry-wide basis, the effect on productivity in any one given installation would not be revolutionary. It is also conceivable that generators could be developed for broad sets of applications. A general-purpose generator for on-line transaction systems seems conceivable today, and the development of one such generator has been reported in the literature [Maynard 1974]. Given the expected requirements for this type of applications, such a generator's impact on development productivity could be substantial.

Assuming that such an approach is practical, it is easy to see how the objectives stated above could be realized. For example, if we look at the difficulties the end user encounters in trying to convey his requirements to the system developer, we can see that this approach would help by allowing the end user and the developer to work as a team in interacting with the generator; conceivably, it could also allow the end user to interact directly with the generator; the applications developer would then be needed only in situations where a new function is required.

One of the basic strengths of this approach is the prestructuring of the application that is gained by the existence of the generator and by the initiative the generator retains in its dialog with the developer. One of the severe weaknesses the industry currently suffers from is that the responsibility for initiative in application structuring is vested in the developer and is, therefore, unique for essentially every occurrence of each application. This has defeated most previous efforts to use standardized modules as building blocks in new applications.

It is apparent that this approach is an extreme departure from current development techniques. If it is, in fact, necessary to make such a substantial departure from current methods (and we believe that it is) in order to solve the programmer productivity problem, then we cannot overemphasize the need for extensive research and development efforts in this area."

Appendix A

Prices for Software Development, Martin Marietta (1976)

"SYSTEMS AT $6....$4....$2....$1.... PER PROGRAM STATEMENT

In-house programming costs $6 to $10 per line including testing. We build customer systems at $6 per line price, performance delivery guaranteed. You don't save much money but you may save time and avoid schedule bottlenecks. You do get flexibility, access to specialist skills and assured results.

Working at your site and in our offices, we deliver programs at $4 per line. We design robust systems: easy to use, test and modify. We pay attention to detail in coding, documenting and testing. We provide a maintenance warranty.

Using off-the-shelf Modular Application Systems (MAS) building bricks, the cost reduces to $2 per program statement, for a system to your exact specification. In some respects, the quality exceeds that of custom built systems: economy of scale permitting unusual refinement. MAS, being built to flex, adapts to changing company needs. So, over the years, the MAS approach can offer a better fit that a custom built system, while avoiding code redundancy.

If only standard MAS modules are used, the resultant system will cost $1 per line. You get all the benefits of MAS: performance proven in over 500 implementations, provision for total integration and honed down operating costs."

Appendix 5


"The Federal Software Exchange Center (FSEC), intended to foster sharing of programs between federal agencies, is now becoming operational under the General Services Administration (GSA) Automated Data and Telecommunications Service.

Outlined in a Federal Property Management Regulation (FPMR 101-32.16) issued last February, FSEC's stated goal is to reduce 'overall costs, time and use of personnel resources for software acquisition and/or development.'

The FPMR calls for the pooling of information of 'common use software' by all federal agencies 'having [DP] facilities, resources or requirements.'

Once gathered, this information is to be maintained in a catalog, published and updated quarterly by the National Technical Information Service.

Agencies will be required to search through the listings of what is available from FSEC before they are allowed to acquire any software from outside sources, according to Chris Bythewood, who has organized the operation at GSA.

Software covered by the exchange is limited to programs written by agency staffs or by outside contractors working for agencies. Explicitly excluded are programs that are classified, proprietary or 'developed with revolving funds' or software 'to which the government does not possess the full rights of ownership,' in the language of the FPMR.

Though developed with federal funding, programs in the FSEC will be considered 'property' and therefore not in the public domain. Only federal agencies will have access to them, Bythewood said, noting however that the status of the software is currently under legal review.

Government 'property' cannot be given away (to a nongovernment user, for example) without specific authorization, he explained. On the other hand, an agency 'giving away' a copy of a software routine still has the routine for its own use 'and hasn't
lost any real property at all - which makes a very awkward situation, logically and legally," he said.

While one part of the FPMR defines what agencies must do to support the creation and maintenance of the FSEC library, another paragraph outlines the expected benefits from use of the exchange and another states what can happen if agencies try to bypass using the center altogether."

Appendix 6
Functions Defined by Welah (1968)

ENGINEERING FUNCTIONS
Determine System Requirements
Determine Logical Design
Determine Physical Design
Determine Organizational Assignments
Determine Procedures
Establish Time Values
Determine Labor Classifications
Coordinate Design Introduction

ADMINISTRATION FUNCTIONS
Performance Determination
Performance Evaluation
Performance Projection
Decision Aids and Simulation
Personnel

SYSTEM CONTROL FUNCTIONS
Message Discrimination and Validation
File Update and Surveillance
Report Data Compiler
Data Services

RESOURCE CONTROL FUNCTIONS
Forecast Requirements
Determine Safety Requirements
Determine Net Requirements
Determine Economic Quantities
Aggregate Economic Planning
Detailed Economic Planning
Determine Dispatch Priority
Select Vendor
Determine Changes From Previous Plans
Material Planning

A MANUFACTURING ENTERPRISE

FIGURE 1.1.1 EXAMPLE OF SYSTEM STRUCTURE
Figure 1.1.2 Evolution of the Programming Systems Product

Figure 1.3.1 The Program Development Cycle

Figure 1.4.1 Software Characteristics Tree
FIGURE 2.3.1 NATIONAL SOFTWARE WORKS

FIGURE 2.1.1 DAIS SOFTWARE DESIGN & VERIFICATION SYSTEM (SDVS)
RUTH, J. C., "DAIS: THE FIRST STEPS" NACCON, ’73 RECORD.

FIGURE 2.4.2 SEF: PROCESSOR VIEW
IA KERNEL SYSTEM

I. Basic User Functions
   - Basic user services
   - Concept definition & modelling
   - User instruction & interrogation

II. Data Structure Operations
    - Basic data structure operations
    - Direct data entry
    - Document/Graphics processing
    - Data migration

III. Procedure Operations
     - Interactive problem solving
     - Basic procedure operations
     - Language processing services
     - Procedure testing
     - Procedure migration

IV. System Management Functions
    - System administration
    - Control program services

TODAY'S SYSTEMS

- TSO, JCL, SPF
- IMS/DC, CUE, IQF
- RS, GPSS, COURSEWRITER, etc.
- DDBA, IMS, GIS
- RPL, ATMS, STAIRS, etc.
- BASIC, APL
- ALGOL, FORTRAN, COBOL, PL/1
- PLAN, VAL, XPL
- TESTRAN, CLEAR, etc.
- SMF, RES, JCL
- Unix, ATOS, CICS, etc.

FIGURE 2.6.1 COMPARABLE FACILITIES IN THE KERNEL SYSTEM AND TODAY'S SYSTEMS

WILSON, M., "THE INFORMATION AUTOMAT." PROCEEDINGS, SHARE XLVII,
Session L303, Montreal, August 15-25, 1976.

FIGURE 3.2.1 SOFTWARE SUPPORT SYSTEM

- 218 -
FIGURE 3.3.1 DATA STRUCTURE OF SELECTED SUBSYSTEM OF SOFTWARE SUPPORT SYSTEM

FIGURE 3.4.1 MAJOR SUBSYSTEM OF SOFTWARE SUPPORT SYSTEM
### Figure 4.3.1 Example of SBA Programs to Produce an Invoice


<table>
<thead>
<tr>
<th>Condition Calculation</th>
<th>Class</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition Calculation</th>
<th>Cust. Rating</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIR</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>GOOD</td>
<td>6%</td>
<td></td>
</tr>
<tr>
<td>EXCELLENT</td>
<td>6%</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.3.2 Summary of SBA Programming Language**

SYSTEM DEVELOPMENT

FUNCTIONAL SPECIFICATIONS

COMPUTER SYSTEM DESIGN

FILE DESIGN AND SPECIFICATION

PROGRAM DESIGN AND SPECIFICATION

PROGRAM CODING

PROGRAM TESTING

FUNCTIONAL SPECIFICATIONS

COMPUTER SYSTEM DESIGN

FILE DESIGN AND SPECIFICATION

PROGRAM DESIGN AND SPECIFICATION

PROGRAM CODING

PROGRAM TESTING

SYNTHESIS APPROACH

FUNCTION SPECIFICATIONS

CONFIGURATIONS

SYNTHESIZE

DEVELOP SOLUTION

SATISFIER LIBRARY

ASSEMBLE SYSTEM

NEED

FILE

DATA

CONDITIONS

PROGRAM SPECIFICATIONS

SYSTEM DEVELOPMENT

HSL/1

FILE DESIGN

AND SPECIFICATION

PROGRAM DESIGN

AND SPECIFICATION

PROGRAM CODING

PROGRAM TESTING

TABLEMASTER

PROCEDURE DIVISION

FIGURE 4.3.3


FIGURE 5.6.1


FIGURE 5.6.1 PROGRAMMING PRODUCTIVITY DEVELOPMENT SYSTEM (PPDS)

JOINT SYSTEM DEVELOPMENT CORPORATION, "AN OUTLINE OF THE PROGRAMMING PRODUCTIVITY DEVELOPMENT SYSTEM (PPDS)," TOKYO, DECEMBER 1976