3. THE CHARACTERISTICS OF LARGE SYSTEMS

L.A. Belady
IBM T.J. Watson Research Center
Yorktown Heights, N.Y. 10598
M.M. Lehman
Imperial College, London

1. Scenario: The Nature of Largeness

In his survey paper on software engineering [8] Barry Boehm observes that "... as we continue to automate many of the processes which control our life-style -- medical equipment, air traffic control, defense systems, personnel records, bank accounts - we continue to trust more and more in the reliable functioning of this proliferating mass of software".

This very brief statement summarizes the intrinsic environmental circumstances that have given rise to the large-program phenomenon and the associated software crisis. Mankind today, as individuals, as nations, as a society places more and more reliance on the mechanization, using computers, of an increasing variety of applications. The latter interface with, control and are controlled by, ever more complex human organizations and activities; and all interact with one another within the operational environments, often in an unpredictable manner [17]. The computing mechanisms are embodied in an increasing variety of equipment of ever greater power and speed. The resultant complexes of machines and their application-oriented and system-oriented programs or software, are conceived, created and maintained by people increasingly remote from the application, from the operational environment, from the mathematical and programming skills demanded of early practitioners, and from the management skills required by the controllers of human activities in the days before automation.

In the early days of computers a programmer, usually a mathematician, scientist or engineer, was presented with a problem. He was able to identify algorithm(s) for its solution. The details of the program subsequently written would depend on his choice of algorithm, on his skill as an analyst and programmer and on the particular set of constraints arising out of the environment in which the program was written and out of that in which it was subsequently to be executed.

The application developer might recognize that in certain circumstances the preferred solution and its program embodiment would fail; would produce a result that was at best less than optimum, at worst incorrect. Failing such programmer perception one would expect that, at best, the machine would detect the circumstances in execution (for example an out-of-bounds number). At worst the computation might complete and the error would be detected subsequently, with consequences that could range from the inconsequential to the very costly. Whatever the case the exceptional was taken care of by human intervention.

With one problem solved the individual or group pursuing some responsibility would encounter other areas ripe for computerization. Thus in appropriate instances (and sometimes in not so appropriate instances) programs would be written with even more of the overall activity becoming computer-based. But the human remained as the link between the separate computations. Still other humans were responsible for administrative tasks such as scheduling the various runs, the allocation of computing and other resources to successive applications.

All those involved in the processes described above soon realized the potential for expansion through encapsulation; the binding of the separate activities into a single larger program. The potential benefit was clear: less human effort (man is a lazy animal); increased speed and cost-effectiveness through the elimination of human intervention which must inevitably involve loss of machine resources; increased reliability (sic) of the machine and of machine processes. So why not let the program take care of all exceptions; why not let the program recognize and sequence the succession of activities; why not let the computer handle the administrative problems of language transformation, resource scheduling and allocation, information storage, communication? Encapsulate as much as possible within a single program structure. Create comprehensive programs. Add bells and whistles. And so it was done. More and more was included. The large program had arrived.

The adjective large as used here, the concept and attribute of largeness that we now develop and characterize, is not intended to reflect the number of instructions or modules comprising a program. Nor do we refer to the size of its documentation, or to the program's resource demand during execution. We do not refer to the program's resource demand during execution. We do not even intend to emphasize the wealth of function contained within it. There is always a level of description at which the function is recognized as an entity, a payroll program, an operating system. The amount of functionality is relative to a level of discourse.

All the above indicators of program size can be expected to increase as a program grows larger in the sense to be described, but the root cause of the characteristics we shall identify is related to the concept variety. A program is large if its code is so varied, so all-embracing that the execution sequence may adapt itself to the potential variety of its operational environment: the specific input, the requested output, and the environmental conditions in the executing system and in the user environment during execution. A program is large if it reflects within itself a variety of human interests and activities. And if it does then it will essentially lie beyond the intellectual grasp of a single individual. It will require an organized group of people to design, implement, maintain and enhance it. And it is the communication between the variety of activities implemented in the program, the communication within the implementing organization, the communication between the implementors and their product and finally the communication between all these and the operational environment that lead to the emergence of the largeness characteristics which we discuss in much of the remainder of this chapter.

2. Phenomenology: Measurement in Software Engineering

The preceding section has related largeness to variety, the degree of largeness to the amount of variety. The variety is that of needs and activities in the real world and their reflection in a program. But the real world is continuously changing. It is evolving. So too are therefore the needs and activities of society. Thus large programs, like all complex systems, must continuously evolve. Alternatively they can only fall into obsolescence and uselessness [5,18].

We discuss the continuous evolution of large programs, perhaps the most fundamental of their characteristics, in a later section. It is introduced here to provide a focus for the data and data interpretation that is first presented to demonstrate that our discussion represents reality and not abstract philosophical musings that have little relevance in the hard-nosed world of applied software engineering.

Moreover, we hope to convincingly demonstrate that the identified characteristics are intrinsic to the use of computers. If this is accepted, two important conclusions follow: firstly, until it can be changed, we must accept the world - in this case the programming environment - as it is and not treat it as we would *like it to be*. Limitations that arise from characteristics we do not fully understand, far less control, must be accepted unless and until they can be changed. Secondly, we can only hope to change and fundamentally improve the software engineering environment - the world we work in and the products we create and maintain - when it is understood; when its characteristics and the causes or mechanisms that underlie them are identified.

This problem of system understanding and mastery is not new. All of the natural sciences have been built and continue to develop on the basis of a common methodology. The universe or system of interest is observed. Gross entities, patterns of behavior, are recognized and global measurements made, until regularities, patterns, trends, invariances are observed. Only then are models and supporting hypotheses created. These in turn form the starting point for a developing theory that relies on prediction, experimentation and further observation for the gradual evolution of the theory. In parallel, there will emerge an experimental and applied science which, in response to societal needs and efforts, leads to an engineering technology.

That is, the initial development of any science is phenomenology-based. It is not in the first place built, as is mathematics, on abstract concepts, axioms, that are gradually developed into a total structure of models that pass tests of reasonableness and elegance. A formal framework and axiomatic theory follow when basics are clear, when it is known what is fundamental or critical, and what is fortuitous. Indeed even mathematics itself has developed from observation of relations in the real world. Thus the study of software engineering too can benefit from phenomenological studies. The topic has arisen because of bitter experience in developing and maintaining large systems. Hence, we are concerned about a more precise characterization of large systems. We must begin by providing some initial data that can set the scene.

3. Some Data: Traditional Indicators

A first indication of the magnitude of the phenomenon may be obtained from data and forecasts on programming expenditure and the programmer population. Table I presents a fairly recent projection of trends in the software industry [34]. It projects an expenditure growth by a factor of two every five years from 2% of the United States GNP in 1970 to over 20% by 1995. Table II from the same source indicates the expected growth in programmer population. Note the implied decline in the number of programmers per installed computer as indicated in column four. We question however whether the projection really takes into account the proliferation of microcomputers or the large program characteristics that form the theme of this chapter. Thus the actual growth of the programmer population may well be larger than that projected in the table. In any event, the magnitude of the educational and organizational problems in the management of programming projects arising from even the indicated growth is clear.

We have been unable to uncover statistics that indicate how expenditure and programming effort have been divided up between small individual programs - whether application or system - and what we shall classify as large programs or program systems.

Table III provides data for a series of systems that are typical of their respective functional areas. The reader will be well aware from his own experience that this small list of "large" programs could be multiplied many hundreds of times. For each system we give a size measure in statements (instructions plus comments) for one release, where each release corresponds to a version of the system as it is made available to the end-user community. The age of the system at release time is measured from first release to the end-users.

Manpower data is notoriously difficult to obtain. Moreover, data definitions and mode of collection differ from organization to organization. Yet it seems desirable to provide some indicator of the effort that goes into software development and maintenance [4]. We found on several systems data pertinent to the number of modules modified between consecutive releases. This number divided by the length (measured in days) of the inter-release interval yields then a convenient normalized measure of effort: the modules handled per day. In earlier publications [5,20] we have shown how maintenance effort remained constant at about 11 modules per day handled, over the life time of the IBM OS/360 (370) operating system. The corresponding figure for DOS (also constant) was about six. A major military stock control system, intermediate in size between OS/360 and DOS/360, for which we have recently been able to study data, experienced a constant module handle rate of about eight per day. For another manufacturer's OMEGA operating system a rate of about 0.8 modules (of a different size) handled per day was observed over a period of four years. Finally, in the banking application system of Table III the rate of making changes appears to have been constant at about 0.75 changes per day over a three-year period. We hypothesise that this essentially stable work-input rate, which in each instance appears not to have changed despite improvements in languages and methodologies used and changes in resources applied, will be found to be an almost universal feature of the programming environment.

With this observation it becomes clear why productivity is so difficult to define or measure in software engineering. It does not represent a meaningful, controllable parameter in the classical, industrial sense, but it is determined by global system and environmental properties that, at present, lie outside our experience, understanding, or control. Nevertheless in Table IV we provide an illustration of the programming rates achieved. The variability as a function of program type is also well illustrated by the data which shows that the programming rate for the structured and relatively simple language processor is some four times as high as it is for the much more complex control programs. We do not here attempt to analyze this data further. Clearly, the methodology of global observations we have outlined in the previous section must be applied systematically over a wide data space to achieve understanding and meaning in the definition, measure and prediction of programmer productivity.

More varied data about a collection of independent programs developed in a large software house is given in Table V. We draw particular attention now to the large volume of documentation and to the varied ratio of pages of documentation per kiloline of code. A similar variability is found in the size of the project as measured by the average number of personnel, and in project duration. The table thus illustrates the difficulty of making general statements about any aspect of programs and the programming process. This impression is reinforced by Table VI which shows the ranges of some project and program parameters for the products of a different software organization involved in contract programming over a period of several years.

The present section has concentrated on providing some raw data that is intended to give the reader a feel for the numbers that arise when large programs and large programming projects are observed and measured. This data does not appear to provide any general measures of the programming process, or of large system characteristics. We now proceed to analyze more systematically the nature of large programs and of the process by which they are created and, as we shall see, continuously maintained and enhanced.

4. Variety: Change and Growth

There does not, at present, exist a general system theory or design methodology for complex systems. There is in fact some doubt whether a complete theory can ever be totally developed or discovered [17]. Nor are there, at least in the realm of computer software, systematic and complete methodologies for system specification and design. Even with the most meticulous requirements analysis, design and implementation process, the product as first released to its users will not, in general, possess precisely those functional characteristics and properties expected or desired in the application and user environment. The systems will require correction and modification after installation.

Moreover, once installed, the user invariably finds it opportune to use the system differently or for a different purpose than that originally conceived. That is, use of the system will suggest functional modification. Meanwhile hardware technology will be developing. Manufacturers are continuously able to develop new or improved processors and devices that offer the opportunity for cost reduction or performance improvements for greater cost-effectiveness. But exploitation of new usage patterns, new application technologies, new hardware, all require the further modification and development of program support. And once operational the modified hardware/software complex can again not be entirely satisfactory, while once again offering still more opportunities for development. So the programs are again changed and the evolutionary cycle goes on. Continuing evolution, the outcome of the mutual stimulation of system and environment, is an intrinsic property of large systems; a property that may be formalized in a *Law of Continuing Change* [5,18,21a]: *A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.*

4.1. Variety Generated by the Desire to Perfect: Continuing Enhancement

The property of continuing evolution is possessed by all complex systems, more particularly all artificial systems [35] created and manufactured by man. Software systems, however, suffer one attribute that complicates the process and leads to a further property, that of continuing modification of the old. Physical systems implemented in hardware, an automobile, an airplane, an atomic reactor evolve through the emergence of newly constructed entities that are redesigned, hopefully improved, versions of older creations. While attempts may be made to modify an existing artifact for experimental purposes, completion of the redesign process leads to the construction of an entirely new instance. The system, much as biological systems, evolves over successive generations. With software systems on the other hand (and to some extent in socio-economic systems such as cities or a transportation system), it is possible and appears more economical, simpler, faster, and in general more expedient, to change and evolve the system gradually through the addition, modification and deletion of code or other system entities. Indeed it may seem impossible to do otherwise.

Modification appears more economical because it requires a smaller immediate capital investment than would re-creation. But this assessment is likely to be based on ignorance or inaccurate assessment of total life cycle costs. It appears simpler because study of a part of a program in its local environment and the paper and pencil (or interactive terminal input) exercise of code modification and augmentation seems to require a relatively small physical and organizational effort. But this is so only if the intellectual (and physical) effort of ensuring completeness and correctness of the change over the entire system and system behavioral spectrum, in itself and in relation to all other changes being made concurrently or being planned, is not taken into account. And it normally is not; perhaps because we do not know how to or perhaps because we do not rate intellectual effort very highly. It appears faster because it is "obviously" quicker to change "a few lines of code" than to re-create an entire system or subsystem in which a major fiscal, temporal and human investment has already been made.

But basic appearances are fallacious. The fallacy stems from the fact that for any individual change (repair, modification or enhancement) these assessments are generally correct. They become tragically wrong when the unending sequence of changes is considered; when the actuality of an evolving usage and maintenance environment is imposed, when it is realized that system structure must degenerate and entropy, as a measure of disorder, increase under a series of, conceptually mostly unconnected, changes.

4.2. Variety Generated by Imperfection: Continuing Maintenance

The preceding sections have discussed the continuing evolution, functional and performance-wise, that a software system undergoes. Definition of system requirements, development of a specification, design and creation of code that implements that design, are all human, intellectual activities not yet subject to the rigor of mathematical analysis, physical laws or the accumulated, ad hoc and pragmatic, but nevertheless definitive, guidelines of engineering practice. Thus the emerging product must inevitably contain faults, design bugs as well as implementation errors. It must, therefore, be validated, either on completion, or repeatedly throughout the entire process, so that faults may be detected and corrected.

Ideally such validation should be based on constructive proofs that guide the design process [11] or on a proof of the identity (in some sense) of the output of each stage of the total process with that of its predecessor stage [16]. However, at the present time applicable techniques have only been developed for relatively simple, self-contained, programs. Extension of such techniques to large multi-function, multi-element systems is, at best, likely to be a slow process.

For the foreseeable future, therefore, validation must continue to rely on inspection [14] and on testing. Effectiveness of the former, however formalized, depends heavily on the system overview, observation and understanding of the inspectors. The effectiveness of the latter will depend on the insight and understanding of the test designers who cannot possibly view the system from all future user perspectives. Moreover, the test designers must cope with a changing, combinatorially large, set of program boundary and environmental conditions, and hence with an impossibly large number of system states and execution trajectories. System validation activity based on these techniques can therefore never expect to locate all faults, can in fact not possibly demonstrate that the system is faultless [12]. In summary, both inspection and testing are likely to be useful in ensuring elements, modules and components, that are relatively clear of localized faults. They become increasingly costly and ineffective in the search for problems stemming from global interactions and dependencies; in ensuring correct system operation.

The faults that are discovered before the product is declared ready for customer delivery will generally be fixed immediately. However, particularly for multi-site, multi-configuration systems, users will, after release, subject the integrated system to configurations and execution patterns to which it has not previously been exposed. Thus new faults will inevitably be discovered and will continue to require fixing.

And the cost of this continuing maintenance is high: Figure 1 summarizes the fraction of programming effort spent in maintenance for a large sample of installations [28] around 1970. An elaborate organization is often required to implement the system change activity. We will discuss this in the next section.

We note that this so-called fix or repair activity is not really repair at all. Hardware physically deteriorates because of wear, corrosion or fatigue. Its repair consists of replacement of one or more components to restore the system to its original state. The elimination of software malfunction, on the other hand, requires a change away from its designed or constructed state. Software repair and maintenance mostly involves redesign which in turn may introduce further error and is very likely to further increase complexity. for the emphasis of a maintenance team will be on speed, on cost minimization, or just simply on obtaining a correct fix. It will not generously include structural maintenance or improvement. And if imperfect repair or structural deterioration is likely when a single fault is fixed, the effect is likely to be compounded when several faults must be cured in the same period possibly by different groups or individuals; possibly concurrently with enhancement and development. Thus inevitably repair activity will be imperfect, will cause the creation of new problems.

4.3. The Result of Continued Evolution: Structural Complexity

Some programs, or parts of the same program system, may be more complex than others, as discussed in Section 3. What is important is that increasing system complexity leads to a regenerative, highly non-linear, increase in the effort and cost of system maintenance and also limits ultimate system growth [4,18,19]. The trend may be summarized in a *Law of Increasing Unstructuredness* (Entropy) [5,18,21a]: *The entropy of a system increases with time, unless specific work is executed to maintain or reduce it.* This, our second law, is analogous to, perhaps even an instance of, the second law of thermodynamics.

For our present purpose its significance is not that complexity increases with age. That is universal experience. What is fundamental to achievement of better software management and minimal life-cycle costs is the recognition that complexity grows unless and until effort is invested in restructuring. Some part of one's resources must be invested in restructuring periodically or continuously. The alternative is to reach such a level of complexity that further evolutionary progress can only be made through re-creation; total abandonment of the system and its replacement by a new system structured, redesigned and implemented to satisfy the most recent operational requirements. In fact, the technological aim must be to achieve the most economic balance between continuous or periodic restructuring and periodic recreation.

Clearly then it is not sufficient for a system to be initially correct. It must remain correct under an unending sequence of changes. To achieve and demonstrate continuing correctness, it is not sufficient that the system be initially well structured. Well structuredness must be maintained despite that sequence of never-ending change. And our studies of analytic models of the programming process [4,5,18,19] indicate that structural maintenance as the system grows is likely to require an ever-increasing proportion of the maintenance resources. Thus maintenance must ultimately become uneconomical, making re-implementation of the system inevitable. Is this point predictable? Is it possible to forecast the point in time at which recreation of the system is required far enough in advance, to ensure completion of the new system before the old one has collapsed, has become unmaintainable?

4.4. An Empirical Study: The Dynamics of Evolution

Rather surprisingly, since every aspect of system development, implementation and maintenance is to some degree planned, and is managed by people for people, the growth patterns of a system as measured by various critical parameters are statistically predictable. This general property is reflected in a third law [5,18,21a]: *Measures of global system attributes may appear stochastic locally in time and space, but statistically are cyclically selfregulating, with identifiable invariances and recognizable long-range trends.*

Recent studies [5,19,20] have reported on the gross growth patterns of a number of very different software systems. Table VII lists some of these differences. The studies demonstrate quantitatively what participants in such projects have long known heuristically, that there is a continuous growth in the functional content, the size, the need for repair and the complexity of each system. Figures 2 show the growth of two systems, measured in number of statements and modules, respectively. Notice the increase in the size trend, and the declining growth rates.

From the same studies, Figures 3 capture the work rates during evolution. The measures, changes made and modules handled are plotted cumulatively as functions of system age to eliminate the effects of release overlap. The slopes of these plots are effectively and unexpectedly constant, implying a constant work rate despite improving methodology, tools and changing resources. The invariance of work input reflects the stabilizing influence of the many organizational feedback loops controlling system evolution.

An approach to the measurement of complexity is by the fraction of total units (i.e. modules) impacted by some unit change: the more units impacted the more diffused the change, the more complex the system. Conversely, in a well-structured system changes tend to remain localized. Figure 4 indicates the trend for two systems studied: increasing spread of changes as the systems age.

An aspect of the interaction between the evolving system and the human organization which drive this evolution is depicted in Figure 5. This shows the linear growth trend of two systems as a function of release sequence numbers. Closer examination shows a cyclic subpattern of increasing period. This cyclicity is the manifestation of the conflict that arises in large system management, between the pressures for an increasing enhancement rate (positive feedback) and the resultant increasing resistance to change, increasing difficulty of the work, as structure and quality, organizational integrity and knowledge decline (negative feedback).

We deduce that an organizational, as distinct from an individual, programming project behaves like a selfstabilizing feedback system. This is of course quite contrary to the view that managers have. They see it as a process whose progress is determined by local and global decisions as these are made. Theoretical models of the process [5,32] suggest that the observed behavior is consistent with our developing view and understanding of the process as a complex, feedback-system-like activity.

That is, gross, historical, software project data of a variety of systems and project attributes can be used to generate project models that represent or measure the evolution process. The models provide statistical invariances, patterns and trends that are interpretable in terms of theoretical models of the programming process. All may be used directly to better plan and control system development, maintenance and enhancement. Equally, their study leads to a deeper understanding of the nature and attributes of the programming process and of the systems that the process produces. The increasing understanding can be applied to direct and guide software engineering practice and the programming process so that the latter may yield higher quality and improved life-cycle properties for programming systems [21a].

4.5. The Life-Cycle Cost Pattern

The previous discussion has been largely system and programming-oriented, directly addressing software system attributes. Quantitative global cost studies have also been undertaken and have produced useful models. These reflect the fact that a very high percentage (50%-90%) of life cycle costs of a large software system may be incurred in post-first-release maintenance and enhancement [1,8,27]. This data provides dramatic confirmation of the reality of the phenomena discussed above. It indicates that the initial assessment of large software systems, the decision to implement or not, must be based on an accurate projection of <u>life-cycle costs</u> not on the estimated cost of development and first implementation. But that in itself is a problem since the expenditure pattern is certainly very non-linear, with shape parameters being a (not well understood) function of the requirements of the system, the implementation and the usage environments. But in this area too, global system observation [27] leads to models that can be most effectively applied in the planning and control environment. Their further development should lead to specifiable and controllable life-cycle characteristics.

5. Large Systems: Complex Interactions

In the preceding section we have concentrated on a description of the global macro-properties of large systems. As the thermodynamicist does in physics or the macro-economist in economics, we have summarized observations, measurement and interpretation of the phenomenology of evolution of the total system and the programming process in their development and maintenance environment.

We have, however, only briefly indicated how large a system needs to be before it can be expected to display the characteristics described. A precise measure is likely to depend at least on the nature and structure of the organization controlling the system, the programming methodology employed, and the nature of the usage environment. But the general indicator mentioned in Section 1 can be deduced from a clearer understanding of the underlying causes of the phenomena described. The basic problem is understanding; understanding the environmental requirements so that the system can be specified, designed and implemented; understanding the design and implementation so that the system may be validated, proven to provide the requirements, all the requirements and preferably nothing but the requirements; understanding the system's properties, capabilities and limitations so that it may be learned and effectively used; understanding the system structure and content so that it may be maintained and enhanced.

5.1. Program Systems

The degree of understandability of a system depends on its structure and its content, the internal interconnectivity between it parts; on its complexity.

More specifically, structure represents the degree to which, and the way in which, the system may be viewed as a set of interconnected subsystems, the way in which the subsystems themselves may be decomposed, and so on. Decomposition aids understanding to the extent by which it enables system behavior to be understood in terms of, or deduced from, the behavior of its constituent parts. The degree to which this is possible depends on the regularity of the structure and the interpretability of the parts as well-defined primitive operators or transformers. Furthermore, the comprehensibility of the system will be heavily dependent on the actual or implied interconnectivity, interaction and dependence between parts, particularly where the related lines of communication deviate from the regular system structure. That is, the understandability and therefore the complexity of even the most well-structured system will depend additionally on the nature and extent of internal, i.e., intra-system, communication.

We may now restrict ourselves to such programs where the structure and internal communication links or dependencies are sufficiently rational to make the system understandable and manageable. As requirements grow and ever larger programs are considered, the point will come where it is judged necessary to employ two (or more) people for program definition, design, implementation and validation. The program is too large to fall fully within the intellectual grasp of a single individual. This might well be the critical characteristic that identifies a large system. The new factor that arises at that stage is of course the need for <u>human communication</u>, intraprocess communication between two (or more) people. And this is a process whose effectiveness will deteriorate rapidly as the number of communicants increases.

A further quantum jump in the complexity of human communication, in the probability of distortion and error, occurs when the number of communicants reaches say eight, when the need for at least two levels of management first emerges. The nature and degree of communication between the members of each of the individual groups is then radically different from that between members of different groups. Comprehension of the total system has certainly slipped from the grasp of any one individual or of an informal team. The system will rapidly become a large system with all its characteristics and problems.

5.2. Program Collections

The preceding discussion has outlined the development of a single program into a large system. Such large systems are met today in the form of operating systems, transaction systems, weapons systems and so on. The components of such systems make frequent use of each other's services and data at execution times. Often they represent the means for controlling, coordinating and exploiting what are otherwise independent hardware apparatus. Under certain circumstances, a set of programs may develop into a large collection of programs. The structure of such a collection, a set of alternative compilers or a number of independent application programs, may be depicted by a wide-span, two-level hierarchy in which a single calling element, a scheduler for example, selects for execution one of a number of alternative programs. Such a collection typically serves a common purpose yet the parts do not, in general, communicate directly with each other while in execution on a machine. In the sense that the word "system" is used, such a collection of non-interacting programs does not form a system. During the development and maintenance phases, loose coupling may exist in the form of decisions about the placement of some sub-capability in one or other program or in a new independent program. But during execution coupling arises only from the calling sequences.

It may, of course, be that this collection of programs is sharing, in some sense, a common data base. In that case the structure and internal dependencies of the latter will act as the communications linkage that causes evolution and degeneration. That is, the total collection of programs with the data base out of which and on which they operate will now form a system. And that system may be expected to demonstrate all the system-like characteristics discussed.

6. The Software Process: Knowledge, Skill and Communication

The most natural fashion by which groups of people collaborate in a programming project [3a] is through decomposition of the total product into separately designable and implementable components [23,26,36]. To form a system, such components must nevertheless interact and communicate. Ideally, the details of protocol, paths, structure and content will be agreed by the designers. In practice, their decisions will be modified during the development process. Additional linkages may be introduced through unintended, often unperceived, side effects. "The evil that these do live after them" [33]. In any event the agreement reached by the collaborators as to the exact details of the interface between their components must be faithfully recorded and strictly adhered to. Subsequent modification must similarly be agreed and recorded.

That is, documentation must be created and updated continuously to record system features, individual design and implementation decisions, the considerations on which they were based, and the details of interfaces between individual system elements. The documentation, ideally a faithful record of the entire process and product, of all internal communication, itself provides a communication link between all process participants and between them and system users. More generally it should provide a permanent, accessible, complete and correct record of innumerable, transient yet possibly significant, interhuman communications. In practice it is of course rare that any of this is done completely and correctly.

6.1. Specialization of Human Activities

At some stage of the process the components, the parts of the system created by separate groups, or by the same group at different times, must be linked together, integrated. As already remarked, no technology yet exists that will guarantee correct functioning of the resultant system. Hence the newly assembled system must be tested. The test responsibility will be delegated to one or another of the original groups, to a new group created out of the original groups, or to an entirely new, perhaps specialist, group. The process changes once again. From being in the domain of a single individual or group from start to finish the process transforms into a sequential, assembly-line-like, activity.

Of course in the very small "large" program the analogy is hardly relevant. But when an organization develops to any size, when systems become really large, expensive and therefore long-lived, division of labor, specialization and the programming analogy of the industrial assembly line may, superficially at least, appear as the most cost effective design and manufacturing process. Architecture, design, programming and coding, data base management documentation, component tests, integration and system test, quality assurance, each activity becomes the domain of a group of specialists. A variety of system elements pass sequentially among them. New programmers, taken on as the more experienced are promoted, must gain knowledge of the system and experience of the process. What better way can there be of gaining that knowledge and experience than assigning to them responsibility for fault fixing, clearing up the many problems reported by users and project teams alike from day to day?

6.2. Product vs. Process Knowledge

There is unfortunately a fundamental fallacy in this approach. The assembly line can work where local processes do not require knowledge of the total product, and the entire process; when individual activities and parts can be totally specified and described, so that elementary operations are essentially independent. Above all a successful line operation requires that total product quality is the summation of the quality with which the individual operations of the process have been performed.

In software, process knowledge relates to methodologies, techniques and tools for specifying, designing, coding, testing and integrating programs, as well as to the planning and management of these activities. Product knowledge relates to the understanding of program elements, structure, algorithms and operation, individually and collectively, and their interactions. It demands a constant awareness of the major objective and requirements, of the multitude of program interactions that occur through common use of program names of objects, such as procedures, tables, labels, variables and so on.

Product and process knowledge and awareness can be obtained by the individual only from the documentation and by word of mouth. Total assimilation is impossible. Hence the assembly line approach cannot be fully effective; is in fact highly fallible. Since no individual can have total knowledge or comprehension, errors are unavoidable and some must remain undiscovered till the product is in regular use.

Assigning repair responsibility to the "greenhorns" is of course the greatest fallacy of all. They cannot, and cannot be expected to have assimilated, the product knowledge, or for that matter the process knowledge, that is essential for effective, structural maintenance, performance maintenance, and functional modification. Clearly repair activity should be the responsibility of those with the maximum system overview and insight. But these are generally the most experienced, the most senior, the highest paid individuals. As such they will often have been promoted into, or out of reach of, the project management. Even if still within the project, they will have architectural or design responsibility, will be working on more advanced system elements, will probably have forgotten much of the detail required for repair.

6.3. The Part-Number Explosion

An additional dimension of complexity in repair arises as follows: Consider modules as basic building blocks, perhaps thousands in number. In a release scheme, there is at any given time only one single valid version for each module. In a multi-installation system, however, repairs (fixes) will often be rejected by users who decide that they are not affected by the fault being fixed. The result is that multiple valid versions of the same generic module evolve; the most recent version, as well as one or more predecessors which will still be actively used at some sites. The predecessors cannot be invalidated to simplify documentation and bookkeeping, since they still exist in the environment against which further errors could be reported and their repair requested [6].

It is easy to demonstrate the explosion of the number of versions. Assume that an error is discovered in module A, in the presence of module B that already exists in at least two versions. It may happen that module A cannot be fixed such that it results in a single new version satisfying systems each having distinct versions of B. Multiple versions of A must therefore be offered even if only one installation reported the error: every customer wants to be prepared against erroneous system behavior even if he ultimately rejects the fix offered. In addition to the redesign process, maintenance thus becomes a complex organizational activity. Additional information must therefore be created in the form of a dependency network that expresses the validity, or the lack of validity, of all fix configurations. The task of creating and updating this network creates an entirely new burden on the fixers and on the installation crews.

6.4. Documentation

The key to system control is system comprehension. One cannot hope to understand the purpose, the mode of functioning and the details of operation of a software system by visual inspection alone, though this might well be possible in a hardware system. One certainly cannot expect understanding of software systems to be discernible from the exquisite level of detail represented by present-day machine level or micro code. Comprehension of the system and its parts requires knowledge of total system objectives, of their partition into individual capabilities or function and of their mapping onto a systems structure and its structural elements. Equally, at least minimal explanation is required of the algorithms used in implementing the system and its subsystems at various levels.

Clearly then a large software system must be accompanied by documentation. Moreover, the documentation must be readily accessible according to the particular needs or interests of the inquirer. And the documentation must keep pace with the changing system, remaining correct and complete.

In practice, of course, this is very difficult. The concept of "self-documentation" of high level programming languages is important but it is insufficient. For machine level languages it does not apply. Thus documentation is necessarily an activity that runs parallel to, and must be interwoven with, the design, system implementation and maintenance activities. As such there is an inherent problem of coordination. When projects begin to lag and to fall behind schedule, when resources run short, the documentation activity, representing as it does a long-term investment that shows no immediate return, being essentially anti-regressive [4,18] in nature, is amongst the first to fall victim to the inevitable axe. Hence divergence between the system, instructional documentation and descriptive documentation is another very typical characteristic of large systems.

6.5. Communication as the Key to Large System Mastery

The analysis so far has identified "communications" as the key in determining the pattern of development of system characteristics. It must be considered at several levels. System-internal communication links join its separate parts and make it a system. Understanding of the system and effectiveness of execution are both heavily dependent on the internal structure as determined by these links. One cannot hope to comprehend the system as a whole unless one is aware of the dependencies and interactions, static and dynamic as determined by both explicit and implicit internal communications. And comprehension of the system as a whole is essential to its effective application and its effective maintenance.

Application and maintenance are essentially the domain of people. These are themselves involved in three further levels of communication. There is the communication between the people that jointly collaborate to build and maintain the system. There is the communication between them and the operators and users of the system. Finally, there is the communication between all of these and the executive management of the producer organization. It is, of course, the latter which controls the ultimate fate of the system [29], together with that of many other artifacts and activities controlled in support of organizational objectives and making a call on organizational resources.

The resultant flow of documentation and verbal communication is enormous and, in general, not clearly structured. Yet for total mastery of the system it must all be integrated and comprehended.

Why is this total comprehension so vital for successful long-range exploitation and control, for continuing control of a system? In general, any interaction with the system, whether for usage or for modification, requires a view of the system as a whole, as an entity. It demands a knowledge of the reaction of the system in its entirety as well as that of each of the parts. The intending user must know and be aware of the total consequence of each system access, and of each of the separate individual responses of which that totality is comprised. Even more strikingly, the individual changing the system in any way must tamper with the code at the lowest level of detail, but be fully aware of the global implication of his action over the entire system.

This need for simultaneous awareness, at both the global and the lowest levels of detail, is brought about by the complex and largely invisible structure of system-communication, the totally unforgiving nature of system execution in the presence of logical error or even imprecision, the rapidity with which the system executes and, therefore, the high probability that any fault, any weakness, will be revealed sooner or later. The need is addressed by system structure and documentation, by system intelligibility, supported by collective human knowledge and understanding of the system.

It is to these areas that we must look for major advances in software engineering, in the development and maintenance of large systems.

6.6. Structure as a Reflection of the Manufacturing Process

Delineation of function and definition of interfaces must occur before autonomously managed groups can begin design and implementation activity. Since the theory of computing system design has not been adequately developed, these definitions and divisions cannot be perfect or complete. In the presence of an evolving environment they cannot remain near perfect or complete. Thus modification must be made as the work progresses, as the emerging system, its function and its structure, is more clearly understood, as the design coalesces and as the system takes shape. Strictly, each such modification should require a total review of all previous decisions. In practice, many modifications appear to be clearly (sic) localizable to remain within the judgment and domain of a single group. Sometimes they clearly cannot be. But in the interest of cost effectiveness, review and negotiation is then limited to those groups most clearly and most directly involved. Implementation of modifications is largely forced into the constraints of the initial structure.

Similarly, when new requirements are identified and subsequently when responsibility for the implementation of supporting code is assigned, management decisions must be based on the existing structure, on the availability of resources, on the existence of localized product knowledge and process experience. In current industrial practice it is largely based on inter-managerial negotiation and bargaining. System structure cannot constantly be reviewed and redesigned to take cognizance of the new features that may well cut right across existing divisions.

Thus gradually, as the design and implementation proceeds, as the system ages, its structure will not only degenerate. Increasingly the relationship to requirements and functional structure will be obscured. The system structure will begin to reflect the organizational structure and process sequence that created it. And this is, of course, not helpful from any point of view. It cannot make the system more understandable, more maintainable, more fault-free. Nor can it be expected to improve system performance.

Possibly the clearest example of this arbitrary structural dichotomy is the division of most organizations into hardware and software groups. Ironically, even with the emergence of new technologies, microprogramming as a replacement for earlier hardware logic design has in many instances been seen as belonging to the hardware dominion. Thus the most basic implementation decision, the selection of an implementation technology, the partitioning of a system into its hard, mushy and soft parts is taken in almost ad hoc fashion at the most primitive stage of system definition.

These approaches may have been correct in the early days of computing systems. It cannot be correct in today's world. And the greatest sufferers are likely to be system changeability, growth flexibility and system performance.

Perhaps the most important contributions to the solution of these problems has, however, already been made. We refer to the extension and generalization of the dual concepts of standard I/O interfaces and of channels. Invention of this latter concepts has made possible the individual design, optimization and system attachment of several generations of new I/O and storage devices. Performance did not suffer because of the incipient potential for device autonomy and system parallelism. We see the generalization of these concepts in the form of a functional channel or Funnel [21], as offering the way to a solution to the problems we have identified as discussed briefly in Section 8.

7. System Behavior: The Optimization Problem

Like most other artifacts, software systems are developed and enhanced with particular objectives in mind. The objectives are often formalized into optimization of system attributes, such as function, capability, cost, reliability, security, size, modifiability, etc. All of these attributes cannot be discussed here, so we single out just one quality indicator: performance. In software this is often considered to require an appropriate balance between execution time of a given subprogram which executes a sequence of functions, and the resource usage required to achieve this execution sequence and speed. The following presents some of the difficulties which software performance optimizers must face.

7.1. System Performance - Execution Dynamics

The factors considered so far have been viewed in relation to the programming process, the development and maintenance dynamics of the large program, its evolution dynamics. When passing to its execution dynamics, its behavior during execution, not surprisingly we observe the consequences of the same pressures, reflection of these same characteristics.

System performance objectives can also not be precisely specified, implemented and achieved in the first instance, and for precisely the same reasons. After all a program is, by its very nature, a complex system of interacting subprograms, subprograms executing on and interacting with shared hardware resources. Performance is not simply predictable [17] and the desired characteristics must be approached via an iterative modification procedure. The endless growth in function, size and complexity tends to degenerate performance. Hence further changes must be undertaken to maintain it. In fact, in the face of application, user and device evolution, performance characteristics normally need to be improved, not just maintained. Thus the execution dynamics of a system, desired and achieved, is a further factor in establishing and maintaining the characteristics we have identified, and itself comprises a further characteristic attribute, evolving performance.

7.2. Local and Global Optimization

The question thus arises as to the extent to which system performance can be optimized, at least relative to requirements and to the environment or at some identified point in time [9].

A problem is immediately apparent. Do we optimize performance under the requirements, environmental conditions and system state as they are now? Then by the time optimization has been completed and implemented, because of continuing evolution, neither system nor environment will be the same. Optimization will turn out not to have been optimization after all. On the other hand, one might attempt to forecast the direction and rate of the various evolutionary processes and optimize to some future expected state. Then the problem may well be that because of changing environmental conditions the expected state is never reached. Alternatively the forecast may become self-fulfilling, achieving optimum performance relative to an anticipated system state that is itself no longer optimum because of unanticipated environmental changes.

The resultant dilemma may appear worse than it is in practice. It is certainly one that may be largely resolved if the optimizers are conscious of the dynamic environment in which they operate. There is, however, a more fundamental constraint on optimization that, in the present state of software engineering, is far more difficult to overcome.

The essential nature of a software system as a set of highly interacting parts has been repeatedly stressed. The parts interact structurally in that they use common program objects. Certain aspects of optimization, storage space minimization for example, dictate maximization of sharing objects. Other interactions come about dynamically during execution. Procedures use each other's services, they share information, they sequentially share hardware. All of these interactions should be taken into account during optimization. In practice, however, such global optimization is very difficult, to say the least. Moreover, designers and programmers have at any given moment an essentially local view of the system in terms of the flow diagram or the code they have in front of them. They have an essentially static view of program flow. A very intensive intellectual effort would be required to convert this to, and assess, the action in terms of the coexistence of concurrent, interacting processes sharing resources. Thus, optimization will unfortunately often tend to be local. And it is well known that local optimization almost invariably leads to global sub-optimization.

There is also a third aspect to the optimization problem in a multirequirement, multifunction system: it is unlikely that all capabilities can be simultaneously optimized. Requirements are likely to be contradictory and while an acceptable compromise must be reached, true optimization may not be meaningful. Even where requirements do not intersect functionally it is difficult if not impossible, as a consequence of data or resource sharing for example, to achieve simultaneous optimization. A data structure that is best for the execution of one function will be suboptimal from the point of view of the other. Optimizing with regard to storage usage is very likely to increase time requirements and vice versa.

Summarizing then we find that optimization for large system performance is a delusion, one that is likely to eat up large amounts of human resources if nevertheless pursued too diligently. In its place, a statement of performance expected and required must form an integral part of the system statement of requirements and of the system specification at its various levels.

8. The Future

The methodological trends stemming from the movements toward structured programming certainly satisfy many of the process and system desiderata arising from the characteristics we have identified for large program systems. The block structure concept first conceived for ALGOL [2], the undesirability of the GOTO construct [13] and its replacement by sequence control structures that are related to the semantic structure of algorithms, the more general movement to high level languages, the channeling of communication via parameter-passing rather than global variables, the single entry-single exit subroutine or procedure, all these are ultimately directed towards increasing the clarity of code structure, its initial intelligibility and therefore its veracity.

But, while necessary, they are not sufficient. If all the rules are understood and observed in their spirit as well as their letter, good structure, healthy code will have been created. But structure must not only be created, it must also be maintained. Structural maintenance, which is strictly antiregressive, bringing no immediate return, must nevertheless form an integral objective and part of the maintenance process.

By and large none of the concepts or techniques mentioned above help in structural maintenance under the conditions encountered in the industrial and commercial world. Even if they can be enforced during program development, the pressured development of fixes for field-discovered faults, or software support for new devices, is very likely to lead to their infringement, and hence system pollution, during maintenance activities.

Moreover, as base systems get ever larger, structure will tend in any case to deteriorate more rapidly during maintenance. Thus the level at which the problem needs be solved is itself rising dramatically. Solutions which may have appeared adequate just a few years ago, now no longer suffice.

One approach to the solution to these problems, program units or components, has been talked about for many years [22] but is only now becoming technically feasible. Moreover, it is only now that manufacturers and users alike recognize the necessity to bypass the problems created by attempts to build and maintain large software systems. The concept is to assemble large software systems from self-contained software units much as hardware systems are configured out of a variety of "black box" units. The recent announcement by IBM [30] of selectible software units highlights the practical emergence of this trend. De facto the announcement implies the abandonment of the large, integrated, system.

The basis of the unitized approach is that defined capabilities or functions are implemented, packaged and offered to users as a unit. Each such unit will (in theory) have been totally tested as an entity against its defining specification. Since in practice the specification will not be absolutely complete, it must also have been tested against other units with which it may interface, against some base system which acts as a central connector for numbers of units or against some standardized and complete interface. Thus we elevate the problem of perception, comprehension and control of the large system to a new and higher level. Software units form the primitives of a new universe of discourse. Their specifications and interface definitions define the semantics and syntax of the design and implementation process. More naively, the process of software system building is then viewed simply as an extension of the standard practice of hardware configuration.

Unfortunately the analogy to hardware configuration breaks down [21]. The critical differences, those that lead directly to the problem whose solution would effectively overcome most, if not all, of the large system characteristics we have identified, relate directly to the two unit-descriptors referred to above, the specification and the interface definition.

For the unitized system to be viable, growable and maintainable over a long period the specification of each and every unit must be complete, relative to stated requirements, correct, i.e., self-compatible, accessible and compatible with the specifications of a significant subset of other units, and ideally with all of them. Each unit must be "pluggable", imposing a known "loading" in terms of its use of system objects. There may be no unidentified side effects, preferably no side effects at all, as a result of unit connection. In Parnas' terms [26] the assumption made by the unit and its environment about others must be completely correct and completely known.

It is one thing to recognize the need for satisfactory specification. It is quite another to achieve it. For the unit concept to work in practice we require a unit specification which is as complete and as accessible as, say, that of a standard bolt or nut. This need was specifically recognized in the emphasis placed at a recent conference on Software Engineering [31] on requirement analysis. But it goes further. The need is not purely for the identification of requirements. Nor will all problems be solved with the development of the specification language that has been the main objective of specification technology for so long. A specification must have structure as well as content. A limited number of candidate structures can perhaps be deduced by identifying system attribute classes and subclasses, and the relationships between them, that together may constitute the specific character of a piece of software. These classes must then be structured and their alternative interrelations formatted into a very small number of specification skeletons that form possible frameworks into which a specification may be developed. After initial exploration of alternatives, one framework must be selected and a specification developed to fill and cover the entire skeleton. The resultant specification can and should be as complete as the framework. It can then form the definition against which the unit itself is developed, validated, modified -- with simultaneous modification of the specification -- marketed, taught, used and maintained. The structured specification form, of course, is the first step in the development of the structured program. It appears to be even more fundamental to the further development of software engineering than is structured programming.

The need for firm and total identification of each unit interface is also clear. The problems raised are more pragmatic. The problem arises from the fact that there is no apparent physical limit to the size of the interface analogous to the surface area and pin limitations arising in a hardware interface. Software communication is volumetric unless constrained by such rules as that of block structure. Even the latter, and certainly any "agreed" interface, can be violated "merely" by a programmer changing the point of declaration of an object, more generally simply by referencing some system object external to his unit, using simple and direct communication as agreed between two programmers. Thus we face two problems: software interfaces are likely to come in an enormous variety of shapes and sizes; even if successfully established, they are very likely to be violated. It is extremely difficult to control them.

The inherent flexibility of the software interface, plus the sociological and managerial problems of maintaining their integrity in the face of three or more generations of managers and programmers brought up without the concept of an impregnable interface standard has led recently to the suggestion that the interface between software units, at least in any one environment, be standardized and implemented in hardware [21]. The Funnel concept is based on a generalization of the channel concept [25] to the point where it is seen as the interface between any two functional units, not just a central processor and an I/O device. It standardizes interfaces by restricting and channeling all communications through hardware links, with the definition of each message on each link being contained within a message header according to a standard grammar.

At first sight such an artificial constraint on inter-software unit communication might appear as a crippling penalty that would seriously degenerate system performance. However, we may recognize that the advancing mini-computer and micro-processor technologies make the whole concept not only feasible but even advantageous. Funnels can be implemented in micro-processor form. Equally, each software unit can execute on its own micro-processor. This in turn leads to a potential for parallel execution which means that the performance limitations of standard hardware interfaces are more than overcome. Thus the large software systems of today will gradually evolve into the distributed systems of tomorrow. But we stress again that such distributed systems cannot become a reality without fundamental solutions, such as those outlined, to the specification and interface problems. Their complete solution, on the other hand, provides the potential for further major functional evolution and growth.

9. Concluding Remarks

The characteristic of continuing evolution that involves growth, maintenance and increasing complexity is intrinsic to the very being of large software systems. The resultant indeterminacies of system state, system function and system capability make them costly to implement, even more costly to maintain and could prove disastrous in certain applications. As a consequence, there is a limit to the size and functional content of software and software controlled systems in their present form. However, the developing technology of micro-processors, together with the concepts of total, structured specification and standard hardware interfaces between self-contained software units is seen as leading to the gradual evolution and future development of large software systems in the form of distributed, highly parallel, systems.

Finally we note that large software systems display all the features characteristic of large systems in general: static and dynamic properties and behavior patterns that are being increasingly discovered and described by an emerging systems science [24]. But they also display peculiar properties that are a direct consequence of the implementation technology -- programming -- used in creating, maintaining and enhancing these systems [21a].

In discussing the characteristics of large systems in the present context, we do not attempt to specifically identify the more general systems properties. It is, however, well worth drawing attention to the contribution that systematic application of systems thinking, the "systems approach", can be expected to make to the future development of software engineering concepts: software systems engineering.

References

1. Aron, J.D.: "The Program Development Process", Addison-Wesley, 1974.
2. Backus, J.W. et al.: "Report on the Algorithmic Language ALGOL 60", CACM, 1960, pp. 299-314.
3. Baker, F.T.: "Structured Programming in a Production Programming Environment", Proceedings of the International Conference on Reliable Software, p. 172, Los Angeles, April 1975.

3a. Belady, L.A. "Staffing Problems in Large Scale Programming", Proceedings of the Infotech State of the Art Conference, "Why Software Projects Fail", April 1978, pp. 4/1-4/12.

4. Belady, L.A. and Lehman, M.M.: "Programming Systems Dynamics, or the Metadynamics of Systems in Maintenance and Growth", IBM Research Report RC3546, September 1971.
5. Belady, L.A. and Lehman, M.M.: "A Model of Large Program Development", IBM Systems Journal, 15, 3, 1976, pp. 225-252.
6. Belady, L.A. and Merlin, P.M.: "Evolving Parts and Relations - A Model of System Families" IBM Research Report RC6677, August, 1977.
7. Black, W.W.: "The Role of Software in Successful Computer Applications". Proceedings of the 2nd International Conference on Software Engineering, San Francisco, Oct, 1976, pp. 201-205.
8. Boehm, B.W.: "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1226-1241.

9. Browne, J.C.: "A Critical Overview of Computer Performance Evaluation", Proceedings of the 2nd International Conference on Software Engineering, San Francisco, Oct. 1976, pp. 138-145.
10. Brooks, F.P.: "The Mythical Man-month", Addison-Wesley Publishing Co., 1975.
11. Dijkstra, E.W.: "Notes on Structured Programming" in "Structured Programming", Academic Press, London, 1972.
12. Dijkstra, E.W.: "The Humble Programmer" ACM Turing Award Lecture 1972, CACM 15, 10, Oct. 1976, pp. 859-866.
13. Dijkstra, E.W.: "GOTO Statement Considered Harmful", CACM, 12, 3, March 1968, p. 147.
14. Fagan, M.I.: "Design and Code Inspections to Reduce Errors in Program Development", IBM System Journal, 15, 3, 1976, pp. 182-211.
15. Oral communication from staff of Fujitsu Corp., Japan, August 1976.
16. Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming", CACM 12, 10, 1969, p. 576.
17. Lehman, M.M.: "Human Thought and Action as an Ingredient of System Behavior", Encyclopedia of Ignorance, Editors: R. Dumcan and M. Weston-Smith, Pergamon Press, London, Nov. 1977, pp. 347-354.
18. Lehman, M.M.: "Programs, Cities and Students - Limits to Growth?" Imperial College of Science and Technology Inaugural Lecture Series, Vol. 9, 1970-74, pp. 211-229.
19. Lehman, M.M. and Parr, F.N.: "Program Evolution and its Impact on Software Engineering", Proceedings of the 2nd International Conference on Software Engineering, San Francisco, Oct. 1976, pp. 350-357.
20. Lehman, M.M. and Parr, F.N.: "Program Evolution Dynamics and its Role in Software Engineering and Project Management", Software Systems Engineering, Proceedings of the Eurocomp Conference, London, Sept. 1977, pp. 393-412.
21. Lehman, M.M.: "Funnel -- a Functional Data Channel", IBM Technical Disclosure Bulletin, 1976. Also Imperial College CCD Report 77/17, July 1977.
21a. Lehman, M.M.: "Laws of Program Evolution - Rules and Tools of Programming Management" Proceedings of Infotech State of the Art Conference, "Why Software Projects Fail", April 1978, pp. 11/1-11/25.
22. McIlroy, M.D. and Boon, C. "The Outlook for Software Components," Infotech State of the Art Report No. 11, "Software Engineering," 1972, pp. 243-252.
23. Myers, G.J.: "Reliable Software Through Composite Design", Petrocelli, N.Y., 1975.
24. We cannot provide here a complete bibliography to (software) systems science. The interested reader may wish to refer to some recently published book in the field and follow references on from there. An example is J.W, Sutherland: "Systems: Analysis, Administration, Architecture", Van Nostrand-Reinhold, N.Y., 1975.

25. Padegs, A: "The Structure of System 360 - Channel Design Considerations", IBM Systems Journal, 3, 2, 1964, pp. 165-180.
26. Parnas, D.L.: "On the criteria to be used in decomposing systems into modules", CACM, Dec. 1972, pp. 1053-1058.
27. Putnam, L.H.: "A Macro Estimating Methodology for Software Development", Proceedings COMPCON 76, 1976.
28. Proceedings of the Symposium on "The High Cost of Software," Monterey, Sept. 1973.
29. "Conference Report, 2nd International Conference on Software Engineering," Computer, Dec. 1976, p. 71.
30. "Selectable Unit Packaging and Distribution", Programming Announcement, IBM DP Division, White Plains, NY, May 1976.
31. Proceedings of the 2nd International Conference on Software Engineering, San Francisco, Oct. 1976.
32. Riordon, J.S.: "An Evolution Dynamics Model", Research Report, Dept. of Systems Engineering, Carleton University, Ottawa, 1976.
33. Shakespeare, W.: "Julius Caesar".
34. "SILT" presentation at SHARE XLM.
35. Simon, H.A.: "The Sciences of the Artificial", MIT Press, Cambridge, Mass., 1969, p. 123.
36. Stevens, W.P., Myers, G.J., and Constantine, L.L.: "Structured Design", IBM Systems Journal, 11, 2, 1974, pp. 115-139.

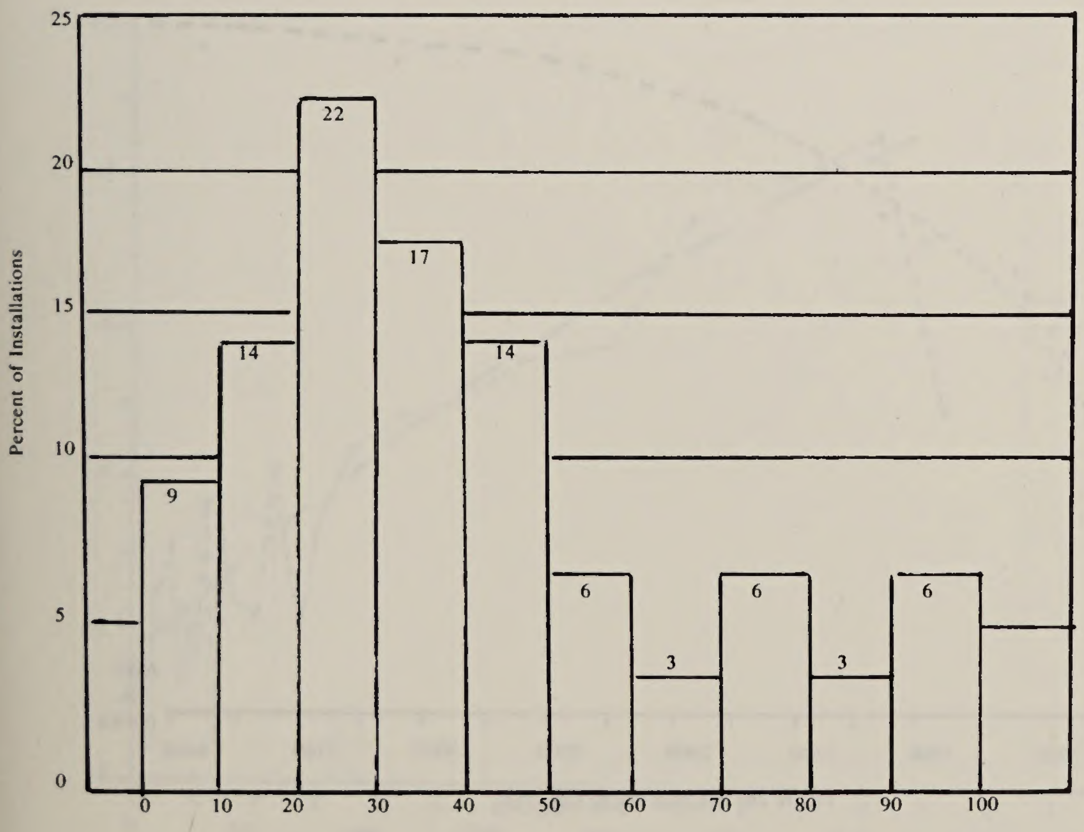

Figure 1: Percent of Time Devoted to Software Maintenance

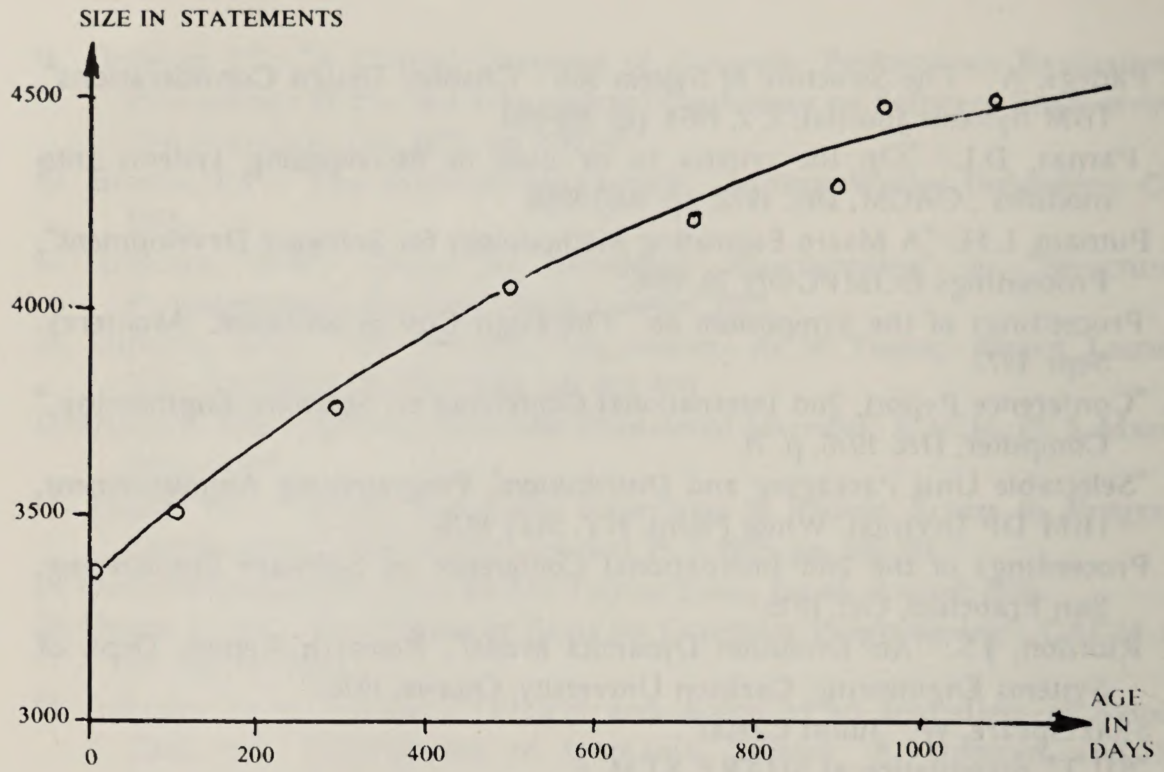

Figure 2a: Growth of the banking system

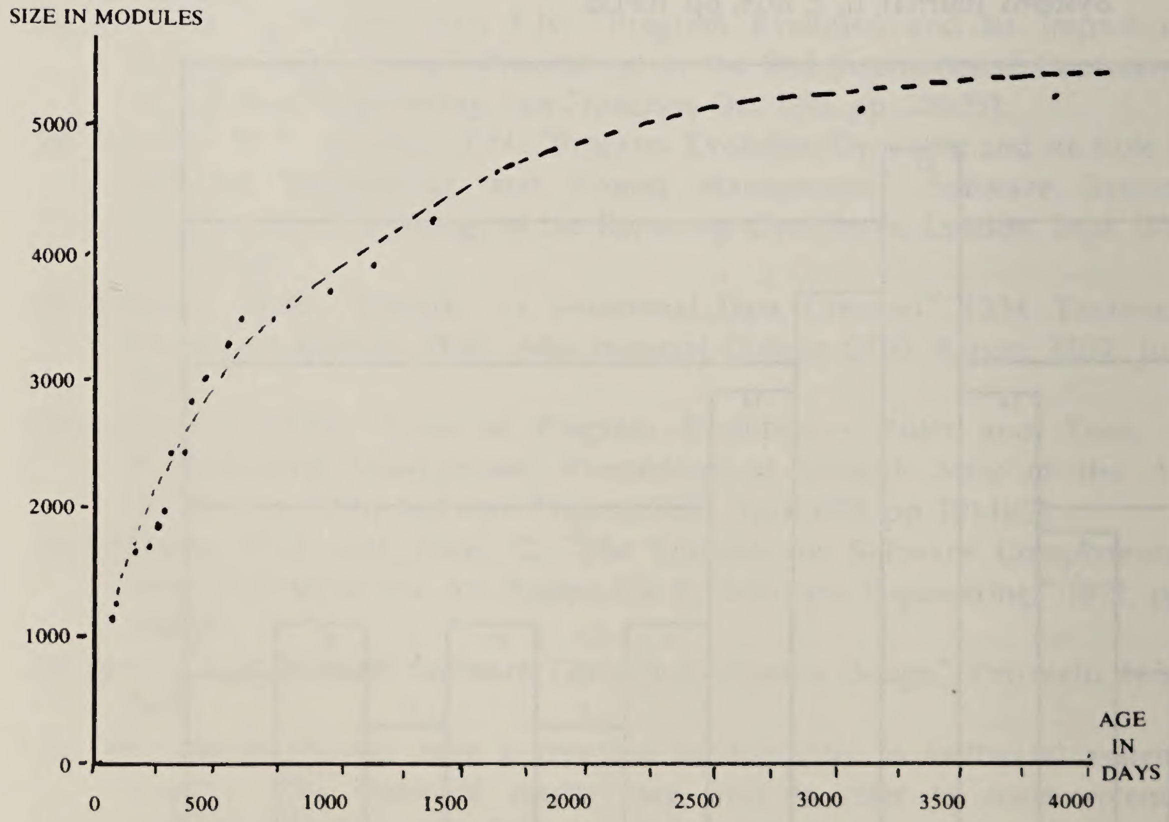

Figure 2b: Growth of OS 360/370

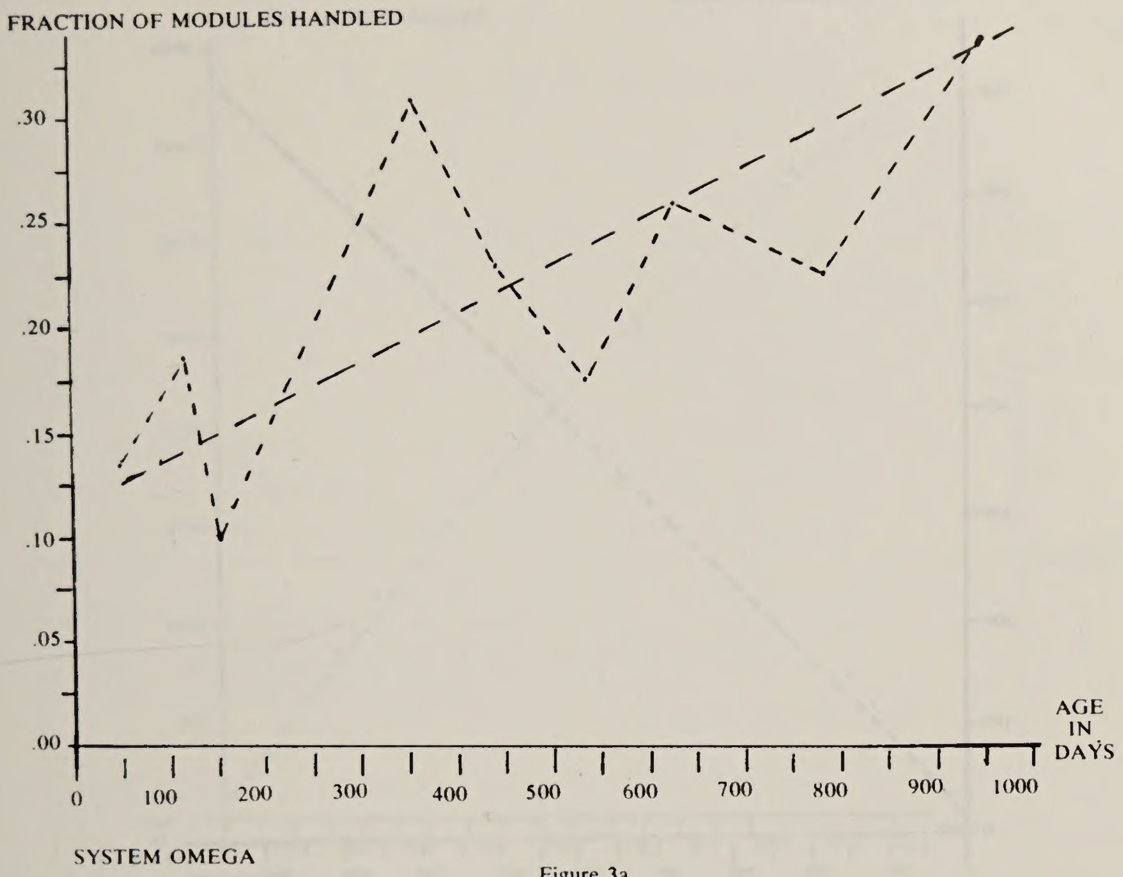

Figure 3a

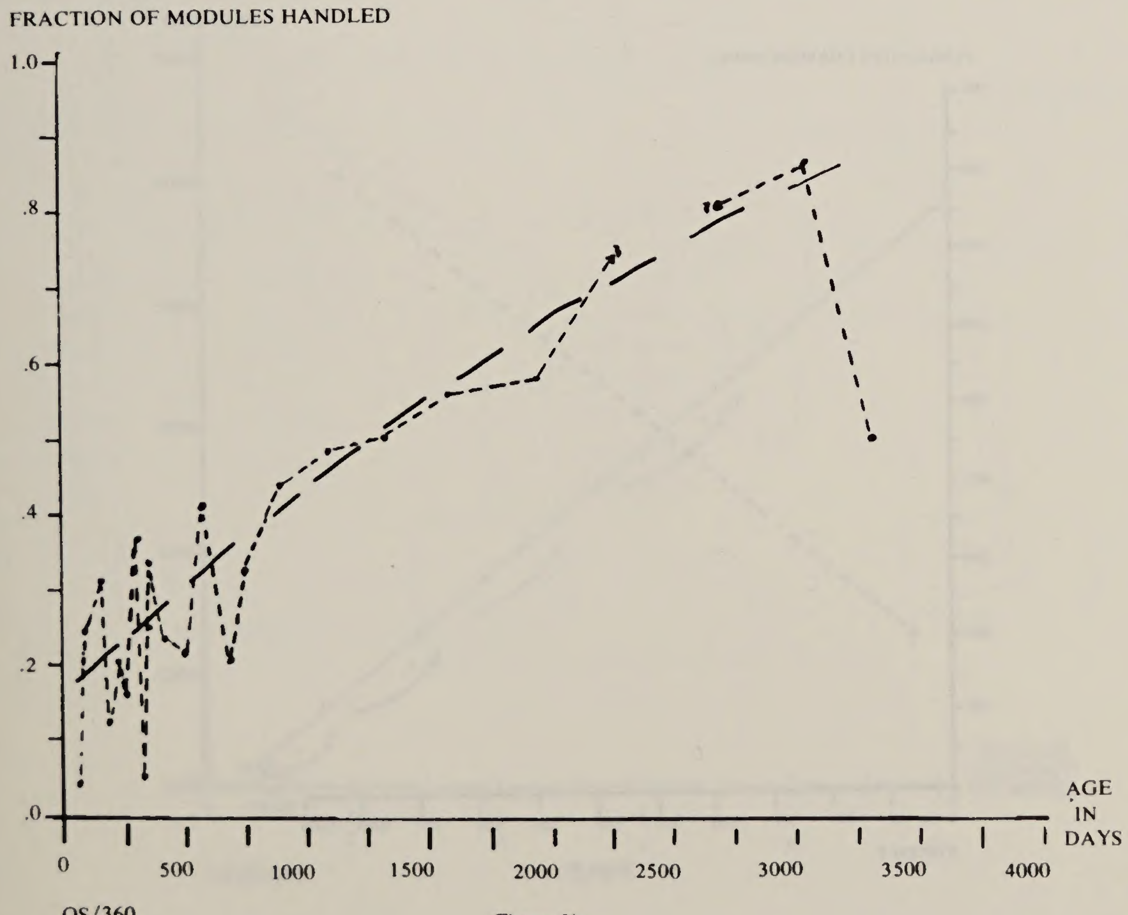

Figure 3b

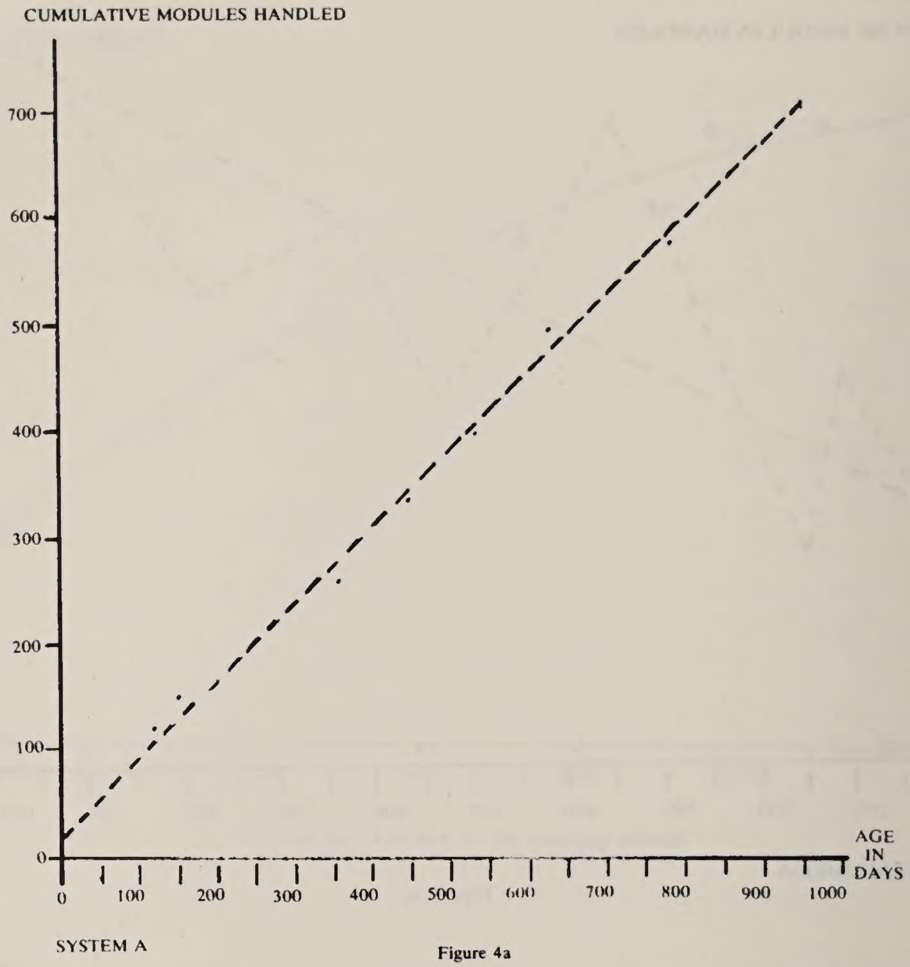

Figure 4a

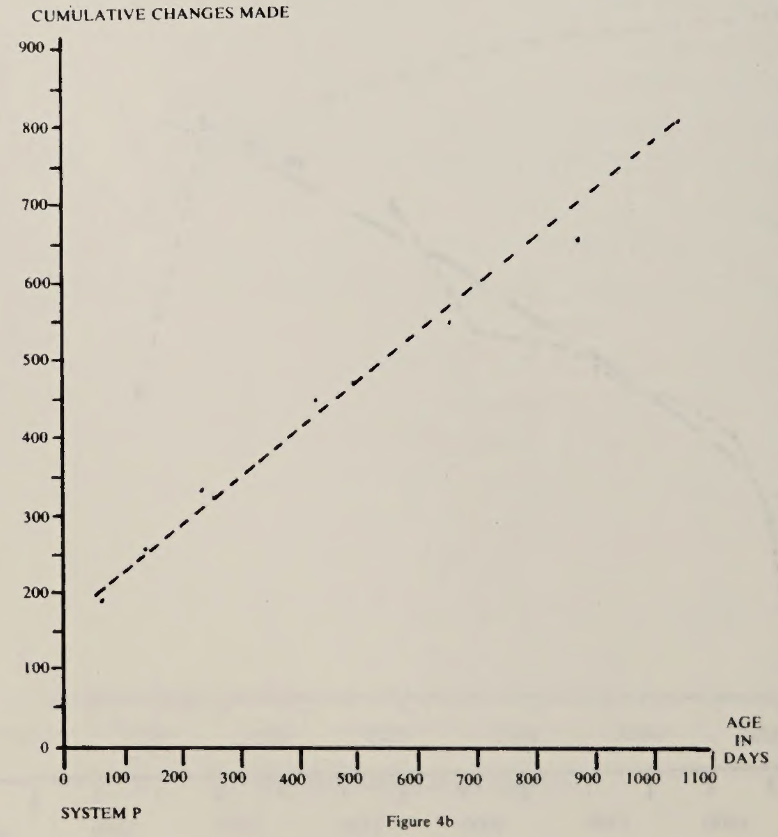

Figure 4b

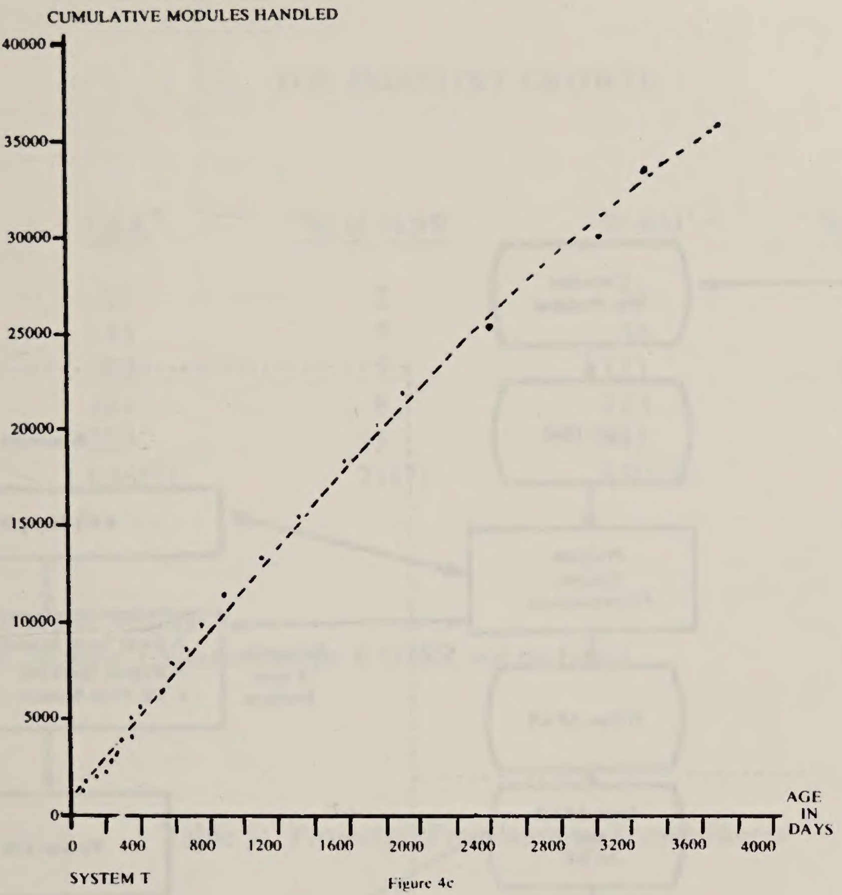

Figure 4c

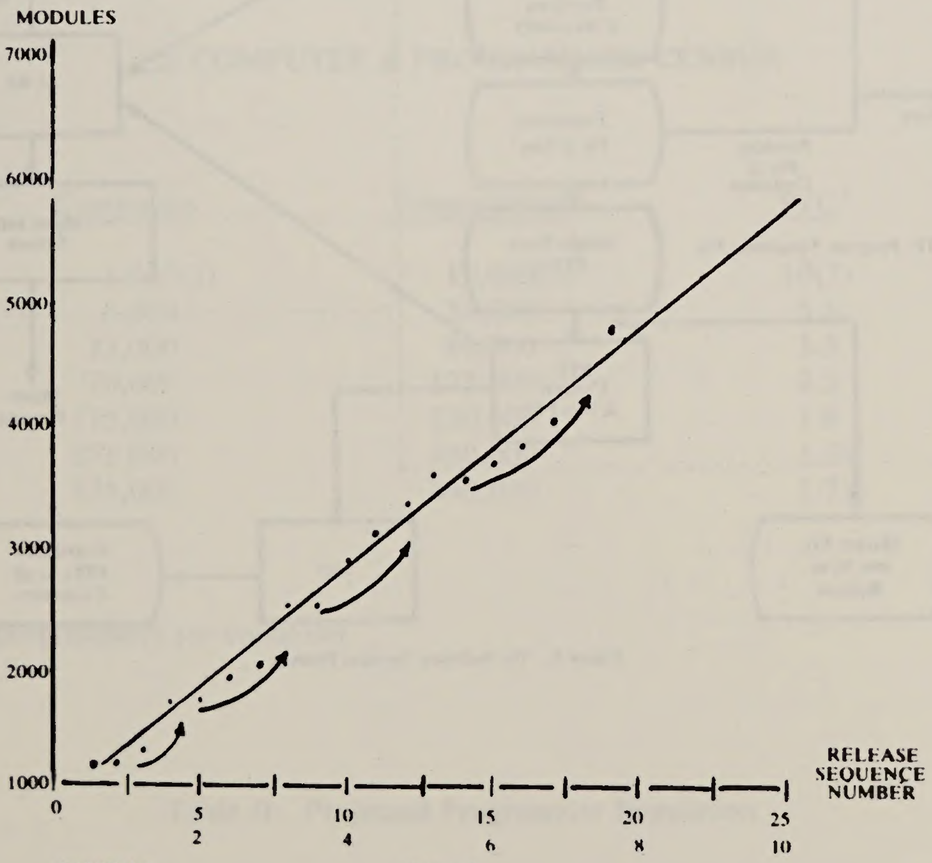

Figure 5

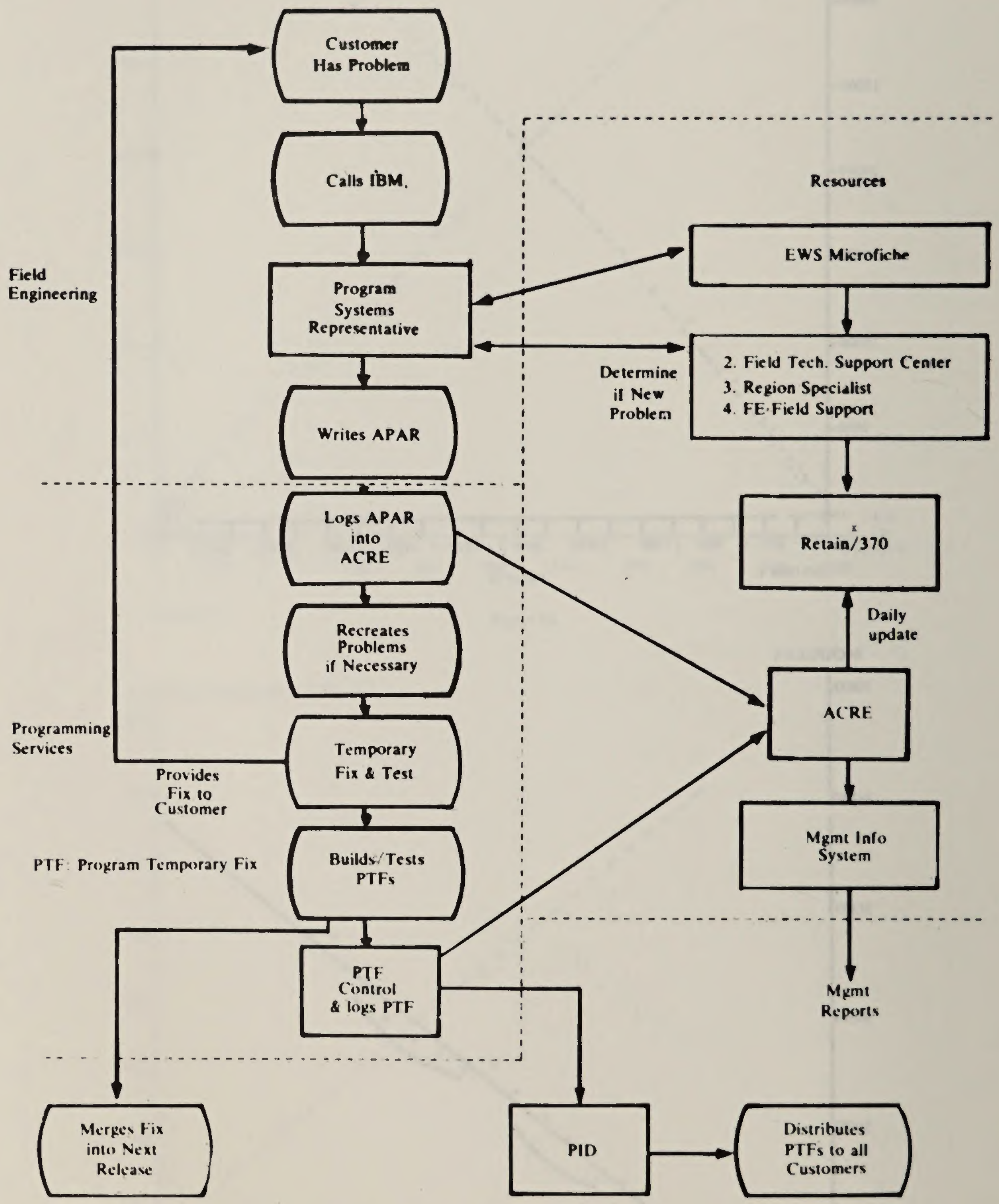

Figure 6: The Software Services Process

D.P. INDUSTRY GROWTH

	USA[1]	% of GNP	World[1,2]	% of GWP[2]
1970	21	2	28	.9
1975	41	3	56	1.4
1980	82	5	111	2.2
1985	164	8	223	3.5
1990	328	13	445	5.6
1995	656(?)	21(?)	890(?)	8.8(?)

[1] 1970 US dollars in billions
[2] Understated because Eastern Europe & USSR not included

Table I: Projected Programming Expenditures

US COMPUTER & PROGRAMMER CENSUS

	Computers	Programmers	P/C[1]
1955	1,000(?)	10,000(?)	10(?)
1960	5,400	30,000	5.6
1965	23,000	80,000	3.5
1970	70,000	175,000	2.5
1975	175,000	320,000	1.8
1980	275,000	480,000	1.74
1985	375,000	640,000	1.71

[1] Number of programmers per computer

Table II: Projected Programmer Population

System	Release #	System age (years)	# (Statements) x10^3	# (Modules)	Language Used
OS 360	21	6.5	3460	6300	Assembly
DOS 360	27	6.0	~900	2300	Assembly
A Banking System	10	3.0	45	-	Algol
Electronic Switching System SPI	20.4	4.0	178	-	Assembly
Electronic Switching System SPC-X3	18	3.0	212	-	Assembly
Building Society Accounting	*	8	150	800	Assembly

Table III: Size indicators of different software systems

*In this case there is only one installation serving 80 users. The release concept is not applied and instead changes are incorporated as developed or tested. About 150 have been incorporated in the system per year over its lifetime.

Project	New Instructions x10^6	Man-months	New Instructions Per Man Mouth	Comments
Apollo Control	1.45	~3800	381	Real Time
Apollo Ground Support	0.53	~1800	294	Simulator
Skylab Control	0.35	~1700	205	Real Time
Skylab Ground Control	1.00	~1100	909	Simulator
A Software House	up to 0.5	up to 12000	-	from a collection of ~ 40 projects
Electronic Switching System	0.166	2500	66	

Table IVa: Programming rates observed on different projects

	Prog. Units	Number of programmers	Years	Man-years	Program words	Words/ man-yr.
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2 1/4	17	38,000	2230
Translator (Data assembler)	15	13	2 1/2	11	25,000	2270

Table IVb. Data from *Bell Labs* indicates productivity differences between problems involving a high degree of variety (the first two are basically control programs with many modules) and those that have better defined specific function. No one is certain how much of the difference is due to complexity, how much to the number of people involved.

PRO- GRAM	— — PRODUCT — —		RESOURCES TOTAL EFFORT (MM)	AVERAGE !# OF PERSONNEL (#)	DURATION (MONTHS)
	DELIV. CODE (SOURCE LINES)	DELIV. DOCUM. (PAGES)			
1	30000	200	77	6	12
2	11164	350	51	6	8
3	17052	450	46	5	9
4	140000	1900	462	15	31
5	47377	78261	241	19	13
6	229000	6100	1665	46	36
7	401099	138016	1022	42	24
8	712362	44000	2176	77	28
9	58540	7650	723	26	28
10	-	187400	186	18	11
11	80990	6000	527	42	12
12	94000	4670	673	16	42
13	76200	6520	-	-	42
14	18775	2000	199	6	32
15	14390	1200	227	13	17
16	35057	60	71	4	19
17	11122	1000	43	5	8
18	6092	427	47	6	8
19	5342	600	14	3	4
20	12000	3000	60	7	8
21	19000	120	50	6	10
22	25271	4500	169	15	12
23	20000	2000	106	8	14
24	12000	1000	57	6	9
25	7000	2000	195	21	9
26	13545	2021	112	7	17
27	14779	400	67	10	7
28	30000	3800	1107	16	68
29	69200	9700	852	24	35
30	486834	41000	11758	174	67
31	220999	15900	2440	40	61
32	57484	8000	-	-	19
33	128330	20880	673	67	10
34	32026	400	136	4	36
35	15363	700	37	5	7
36	4747	200	10	3	3
37	99000	8800	-	-	47

Table V: Statistics for Programs developed by a large software house

Type of Data	Range
(1) Number of Machine Language Instructions	15,000 to 3,600,000
(2) Number of Cumulative Trouble Reports	1 to 1685
(3) Number of Releases	1 to 7
(4) Time Span of Releases	6 to 80 Months From First to Last Release
(5) Average Error Rate: Errors Per Month Per 1,000 Instructions	.016 to .276
(6) Length of Time in Test Mode	1 to 5 Months Per Release
(7) Duration of Use Per Release	1 to 31 Months
(8) Percent of Compiled Code	0 to 100
(9) Percent of Assembler Code	100 to 0
(10) Percent of Code Increase/Decrease From Release to Release	-27 to +67
(11) Percentage of System Disabling Errors	0 to 20
(12) Number of Users	1 to over 1,000

Table VI: Variation for Program Statistics within a Software Organization

System	Srce.	Type	Language	Purpose	Size Inst.	Users	Hdwr.	Configuration	Impl. Env.	Mang. Ctrl.	Age* Days	Rel. Seq. No.
Omega	Manufact. urer	Transaction O S	Assembly	Limited	?	25	Var.	Var.	Prg. Centre	Conc.	1000	10
Bank. Syst.	Bank	DB syst	Algol	Single	45k (H.L.)	1	Fxd.	Single	One Group	Unifd.	1000	11
OS/-360 VS2	Manu. fact-urer	O S	Assembly	Universal	>2M (L.L.)	>1000	Range	Mult.	Distr.	Distr.	3500	23

*Age refers to period over which data is available, the first two systems have a prehistory.

Table VII: Three large systems whose evolution has been studied