Software engineering : methods and techniques

Eindhoven University of Technology
Computing Centre Note 1981/3

SOFTWARE ENGINEERING:
METHODS AND TECHNIQUES

Ir. J.J. van Amstel

februari 1981

0. The evaluation of methods.

During the last 10 years there has been a growing interest in the methodological aspects of software development and many new methods have been proposed for improving the programming process. A new discipline called *software engineering* has been created in the academic field with the goal of providing a scientific base to the entire endeavour.

Today, every programming manager is confronted with many alternate methods for doing the same job and receives conflicting advise depending on who he is talking to. In fact, there is even no agreement on what belongs to the so-called modern methods and what does not.

The properties to be considered for the evalutation of methods are:

- applicability
- affinity
- adaptability
- precision
- effectiveness and
- cost.

The most basic property of a method is its *applicability*. In the same sense that no mathematical algorithm is able to compute all possible functions, there is no universal method that solves all problems and tasks. Sometimes one gets the impression that people believe that, in the case of programming, they have found or should be able to find this universal method. Whoever did an analysis of programming errors soon will find out that the more errors he studies, the more different reasons for errors become visible. For every different error reason a different method will apply, either for its prevention of for its detection. It is symptomatic of the entire software engineering literature that very little is being said as to which specific task within a project and which specific problem is being addressed by a certain method. Many methods also make the assumption that the normal programming environment is one where all programs are written from scratch.

There is a relationship between tasks in a project, applicable methods, and tools supporting the individual methods. Some methods may have a very narrow and specific application only, others may apply to a whole range of problems and tasks.

Any two methods may be completely unrelated to each other or they may

- form some close relationship. We use the term *affinity* to express which are the other methods that a give method can coexist or interact with. Sometimes methods support each other, they form a compatible chain in the sense that the output of one can be used as input for the other, or the strengths of one method compensates for the weakness of another. Examples that complement each other are structured programming and Nassi-Shneiderman charts, or HIPO and Pseudocode.
- The *adaptability* of a method describes the ease with which a method can be adjusted to an environment different than the one it was originally conceived for. As an example, the benefits of the chief programmer team organisation can also be obtained if only partial aspects of the Baker/Mills ideas are actually implemented. This also reduces the level of commitment that is required to introduce a method and makes it easier to back-out again.
- The *precision* of a method indicates how predictable its results are. Most methods are more or less imprecise. They give general guidelines only. For several methods major subaspects are precise, which means that part of the method can be formalized and hence mechanized. As an example, in the case of program proving, whenever preconditions and postconditions have been found, the correctness of a given program may be demonstrated mechanically, i.e. with the aid of a verifier program. In most cases, however, programs can only support minor aspects of the method, e.g. editing of texts or diagrams. The practioner, of course, has a strong preference for precise methods provided they have been mechanized. This makes their results much less dependent on the skills and motivations of the persons involved.

- The property that is most frequently talked about is *effectiveness*. It designates the degree to which the problem or task in question is being solved. There are different and conflicting dimensions when effectiveness should be expressed. For a programming project there are usually three goals that count: product quality, development efficiency and adherence to schedules. During the development process the product quality is influenced by the rate of error occurrence and detection. Methods that prevent the occurrence of errors or facilitate their detection contribute positively to the quality goal. Very little quantitative data is available for any of the methods to express their effectiveness in this respect. An ineffective method may only scratch the surface of a problem, while an effective method may solve the problem completely. Methods with a broad applicability may be very effective for one problem but less effective for another.
- The final property to be evaluated is *cost*. With every method, certain efforts are associated for introducing and using it. These may be reflected in actual expenses, e.g. for tools or for people and machine time. It may also show up in terms of the skills that are needed or the flexibility that may be sacrificed. Whenever effectiveness is measured in terms of development efficiency or productivity, the costs for a new method are usually netted out immediately against the savings. If the goal of the methods is something else, extra costs accrue which are due to the particular method. It is astonishing how little has been published in this respect.

  The evaluation of methods can be done at different levels. Four levels will be considered.

  The first level would be a *study of the proponents' claims* This can be done based on the appropriate literature. This will give some information on applicability, maybe also some on effectiveness, but probably very little on adaptability and cost.

  The next level would be to *interrogate current users*. This is sometimes done by user organisations. The additional information gained maybe in the areas of adaptability and cost. It may also give more reliable data on effectiveness.

The third level would be a *trial period* within the user organisation. For this, a pilot project would be selected. It should be as much as possible a project that in all other aspects is comparable with previous projects. Ideally the same project should be carried out, using different methods thus giving equivalent control information. In practice, this is hardly possible. It is very difficult to exclude all other parameters that may influence the result. Another risk in a pilot project is the socalled Hawthorne effect. Whoever has been selected from multiple candidates for a certain experiment has a tendency to produce positive results.
The fourth level of evaluation would be one where we try to *understand why* a certain method works and why it produces certain results. This would require a "basic understanding of the cognitive processes and problems involved" in programming. This today is still an area of research in psychology, group dynamics and human problems solving theory. It is doubtful whether this level of evaluation is both necessary and possible.

Summary.
Software engineering as a discipline should not be satisfied by just producing new methods for the development of programs, nor should those methods be limited to the case thet programs are built from scratch. What the practioner needs are methods that allow to make use of existing products. This implies that we know how to evaluate the products that are offered.
It may be a goal of the industry to lower the cost of programs. This goal is different from lowering the cost of programming. The road that leads to this goal certainly involves taking a different attitude towards commercial software packages.
Because of the number and variety of methods proposed for the programming process a consistent methodology for the evaluation of methods is needed. This would give a framework in which the scarce results of empirical studies could be classified. As long as no agreement exists between computer scientists as to how our methods should be evaluated, we have not reached the status of a mature engineering field.

1. Software Development Methodology.

1.1. Software development methodology emerges as a mean to cope with the growth in size and compexity of software. It can be considered as an orderly way of developing software, which consists of a series of well defined steps. The most important steps in software development methodology are: *requirement specification*, *design*, *implementation* and *validation*. At each of these steps, the properties of *completeness*, *consistency*, *inambiguity*, as well as *invariance*, must be validated before the next step can be started. Moreover, each step is supported by analysis techniques, design aids, and sofware tools. The methodology provides guidelines for design and programming, and reduces validation and testing by constructive analysis.

1.2. *Requirement* can be defined as statements of constraints on the entity, while *specification* is a structure form of requirements. Specification is an important part of software development because proper specification of the system can prevent and detect some (design) errors by early analysis and systematic decomposition. Moreover, it can partically validate the system throughout the development process, independent of implementation and hardware by maintaining an abstract model of the system and performing analyses on this data base. Further, specification can aid the software development: It can help packaging according to logical properties, performance requirements and availability in software design. It leads to modularization, as well as complete and unambiguous interfaces in implementation. Also, it can assist interface checking in integration and test case generation in evaluation. Lastly, specification can monitor system evolution by locating modules that require to be changed and reassure that the new system will perform properly.

The specification methods can be classified according to the phase in the software development cycle in which the method is most applicable into: *specification of objectives and needs*, *specification of system requirements*, *specification of data-processing subsystem/ software requirements*, and *specification of software process design requirements*. The specification method of system objectives and needs are usually in free form English. It is not formalized as well as difficult to formalize. There are some semantic models in this area, specifically for software requirement analysis. There are also not much formal methods in system requirement. Techniques for specification of the earlier phase (system objectives and needs) and the later phase (software requirements) seem to be applicable in this area. In software requirements, the specification methods are numerous and can be classified according to the aspects of the system being specified into functional specification, concurrent process synchronization, data flow specification, control flow specification and performance specification. In software process design requirements, specifications includes algorithm specification, data type specification, as well as concurrency specification.
Although some techniques are available in requirement specification, there is no standard or formal way of expressing them, and the software methodology in this area is not well understood.

Program design involves the creation of forms that satisfy certain predefined functionality with maximum economy and efficiency. The many design methodologies in existence today include top-down, bottom-up, composite/structured design, Jackson's method and SADT. Top-down methodology is the prevalent methodology which involves reducing a large complex problem by decomposition and partitioning into modules which are simpler to solve. Through a series of abstractions and elaborations, stepwise refinement, preserving uniform control structures, modularized code results in a program which is easier to understand, test and verify.

In order to insure the quality of critical large scale software systems, extensive validation and testing procedures are required before they are implemented in their real operational environment. Because exhaustive testing is not possible on implemented code, intelligent testing strategies involve the use of test case generation. At present there is no acceptable theory of program testing. *Program validation* involves the test and evolution of software aimed at ensuring the compliance with the function performance and interface requirements. The validation task can be quite extensive and certain constrictive techniques like top-down design, structured programming and imposing restrictions on the programmer's freedom can ease the process considerably. Program validation include proving methods and software evaluation methods. Verification-proving techniques may be approached formally using predicate calculus or informally forming logical assertions about allowable ranges of variables, inadmissable states and relations.

2. General System Issues.

2.1. *Decomposition.*

Decomposition is the process of dividing a system or a process into several levels of subsystems and subprocesses based on certain criteria. Decomposition is needed for orderly design of complex systems. It is used to identify tightly coupled processes, to minimize the interactions among processes and to provide a clean interface between various processes and better resource utilization. There are two approaches in decomposition, namely, functional decomposition and attribute decompositon.

2.2. *Partitioning.*

Partitioning can be defined as the process of dividing the decomposed subsystems or subprocesses into different modules. Partitions must be well-defined, i.e., it must be consistent, complete, unambigious, testable and easily integrated. Also, several operational requirements such as resource allocation, performance, reliability have to be met. Partitioning can be achieved by exhaustive partitioning, natural ways partitioning (physical or logical) or artifical ways (centralized or distributed).

2.3. *Specification Languages.*

A precise statement of specifications is required to design a large secure system. The objectives of specification languages are to prevent and detect some design errors, to partially validate the system throughout the development process, to aid the software development and to monitor system evolution. A good specification language for software requirements should precisely, completely and unambiguously state the following aspects: 1) functional specification, 2) concurrent processes synchronization, 3) data flow specification, 4) control flow specification and 5) performance specification.

2.4. *Validation.*

Validation is the process of ensuring that the product satisfies all originating requirements. Properties validated include completeness, consistence, ambiguity and invariance. Some constructive techniques in the design process are very helpful for validation, for instance, top-down design, structured programming, decomposition and partitioning methodologies. Verification techniques include checking input/output consistency, logical consistency and proving some specific properties like deadlock-free and security criteria. These can be either verified by symbolic execution or proved by theorem. Difficulties in this area result from complexity of the program, lack of up-to-date documentation, hardness in simulating execution time environment and imprecise or incomplete system specification.

2.5. *Testing.*

Testing can be classified into simulation testing or actual system testing. There are several phases in testing: program analysis, test data selection, formulation of testing strategy, environment simulation, actual execution of programs and evaluation of the testing process. Since exhaustive testing is not possible on implemented codes, a good approach is to employ intelligent testing by test case generation. Today there is no acceptable theory of testing. For future works in testing, we may see more emphasis on constructive approaches and combination of proving and testing.

2.6. *Complexity.*

Time complexity of an algorithm is the measure of its execution time as a function of its input size. Computational complexity of a problem could be polynomial time, exponential time or NP-complete (i.e., not likely to find a deterministic polynomial algorithm to solve the problem). For NP-complete problems, some heuristic methods have to be used to find the solution close to optimality. In additional to time complexity, processor and memory requirements of an algorithm have also to be dealt with.

2.7. *Modeling.*

Modeling is a formal abstraction and generalization for the description of some objects. A model is independent of representation and implementation. The use of modeling can make problems easier to understand and analyze. Moreover results obtained by studying the models can be reused. There are two types of modeling, namely simulation and analytic models. A number of different models can be constructed from different points of view and have different purposes or contain different amounts of detail. The various models of a system correspond to different ways of partitioning it into components, of representing the interactions among them and with the system's environment. A lot of models have been used and analyzed in computer system design, such as Turing machines, finite state machines, directed graph models, Petri-net models etc.

2.8. *Concurrency.*

Processes are concurrent if their executions could overlap in time. In a multiprocessor or distributed system, the executions of concurrent processes overlap in time. But in a uniprocessor system, concurrent processes can only execute in interleaved time slots. However, the logical problems turn out to be the same in both cases. We need some mechanisms to deal with synchronization of interactions to resolve or prevent deadlocks of resource allocation, etc. Especially, concurrency among processes has to be precisely identified and represented in specification languages.

3. Software Requirements and Specifications: Status and Perspectives.

3.1. Introduction.

3.1.1. Motivations and Software Specification Problems.

Data processing systems are being entrusted with increasingly complex and critical functions. The *cost of software* in these systems is becoming dominant, being more than 50% currently and expected to rise rapidly.

The main problem with software systems is their *complexity*. Lack of a systematic design and development methodology to handle the complexity has given rise to high costs, slippage of production schedules and inadequate operation and maintenance.

Among the various classes of erros that have been documented in studies of software systems, *design errors* have been found to account for a large fraction of the total. Design errors are those errors that involve changes in the design specification or a reinterpretation of it.

Design errors orginate in the phase of system design known as requirement engineering. The activities in this phase are concerned with defining the functional needs, performance and other requirements. Many problems may originate in this phase, e.g., inconsistent requirements, ambigiously expressed requirements, incomplete statements, etc.

3.1.2. Importance of Specification Methodology Development.

The main objective of a specification methodology is to tackle the above-mentioned problems by providing a means to precisely state the system requirements and provide analytic procedures to check their consistency and completeness. In addition it:

a) facilitates coordination of programmers by providing precise specifications
b) supports unit and integration testing with testing specifications
c) reduces maintenance costs by providing monitoring functions and provides traceability for problems which develop during the operational phase.

3.2. Multiphased Development.

In a large and complex design situation, the transformation of the problem needs to the final product takes on a number of distinct forms which progress from the general to the specific, from the abstract to the concrete and from the aggregate to the detailed. The form resulting from one phase of the transformation is regarded as the specifications or requirements for the next. Within the context of this multiphased development the following issues arise:

a) Specification Languages: Because of differences among the development phases, a single language does not suffice to express all forms. The languages must be easy to use and comprehend and adapted to the expression of the class of objects involved.

b) Validation: To support validation the forms must be expressed in a precise notation with unique interpretations for the expressions used. We must be able to show that each requirement in one form is satisfied by some combination of the specifications in the next.

c) Feasibility: It is desirable to show that each form is feasible so that the specification process does not have to be backed up. At present, the only techniques available are prototype testing and simulation.

3.3. Software System Life Cycle.

Institutionalized software development projects are usually divided into phases similar to those shown in Figure 1. We describe a typical and somewhat simplified software life cycle below to orient the later use of terminology.

The least disputable aspects of the cycle is that it begins with the conception of some needs to be fulfilled, and there are feelings and indications that a significant portion of the solution is using the information processing power of modern computer systems. It undergoes the process of development, and then the operational product is used in the actual environment, simultaneously modified, and adapted to new needs.

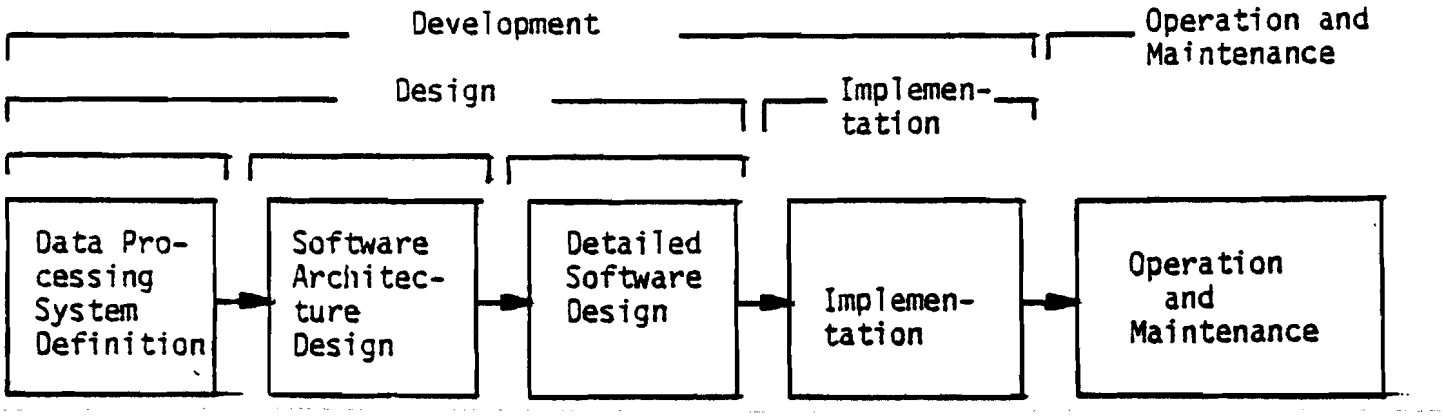

Figure 1. Software System Life Cycle

The development phase is subdivided into design and implementation. The end of design is generally regarded as the end of the more creative activities, and the rest (implementation) is a relatively straightforward procedure. In software development, a complete design is the definition of all program modules on a specific data processor, plus the support documentation.

The design process is further partitioned into three subphases of Data Processing System Definition, Software Architecture Design, and Detailed Software Design. The key phases and forms are identified in Figure 2.

This figures shows only the essence of the process. In the development of a large system, the process can be much more complicated. There will be inevitable feedback from one phase back to previous phases, when errors and other difficulties are discovered. Another deviation from the simple scheme is in the incremental development approach for some systems. (A part of the system is built to the operational stages, and then some other functions and performance capabilities are added onto the existing operational parts.) This can be viewed as a series of the above basic process.

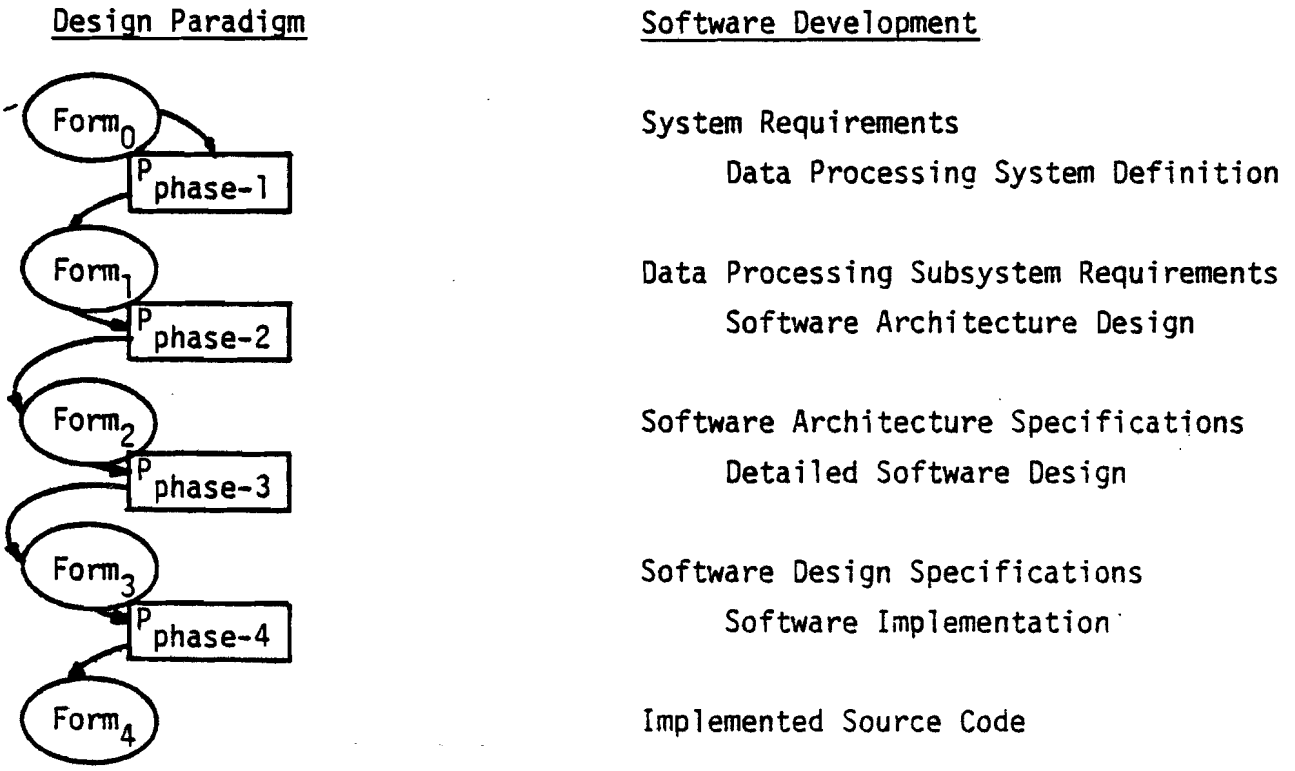

Figure 2. Software Life Cycle

The following paragraphs discuss in more detail the major activities and design decision in each of the development phases.

3.3.1. Data Processing System Definition.

Data processing system definition consists of analyzing and decomposing the system needs and objectives into system engineering terms. System objectives can be very vague, and the fulfillment may not be objectively verfiable. For instance, to improve national defense can be the overall objective of a missile defense system, and to increase corporate profitability can be the objective of the real-time management information system of a manufacturing firm. The objectives must be decomposed, elaborated, and stated in precise terms.

Furthermore, in most system developments we are considering, the data processing functions only constitute a subsystem of a larger complex involving other hardware facilities. In these situations, the total system requirements must be analyzed and part of them allocated to the data processing subsystem. The minimal activities to be performed are: 1) identification of all the entities interacting between the system and its environment; 2) statements of the expected functional responses to system stimuli; 3) identification of system performance parameters and the expected system performance in terms of these paramters (e.g., response time, cost); and 4) specific characteristics of the system (e.g., flexibility) formulated formally in terms of the system elements.

3.3.2. Software Architecture Design and Detailed Software Design.

Software system architecture design has much in common with the later phase of detailed software design. The general design process, as well as software system design, has been the focus of much research. An extensive discussion of the subject is beyond our scope here. We know of no better method to design and develop large system other than the divide-and-conquer and hierarchical structuring approaches (these include the top-down design, stepwise refinement, abstract machine layers, etc.). Thus in the process hierarchy approach, a system is organized as a sequence of hierarchical levels of processes. In each level we observe a group of interacting processes--each of which is accomplished by yet another group at a lower level (Figure 3).

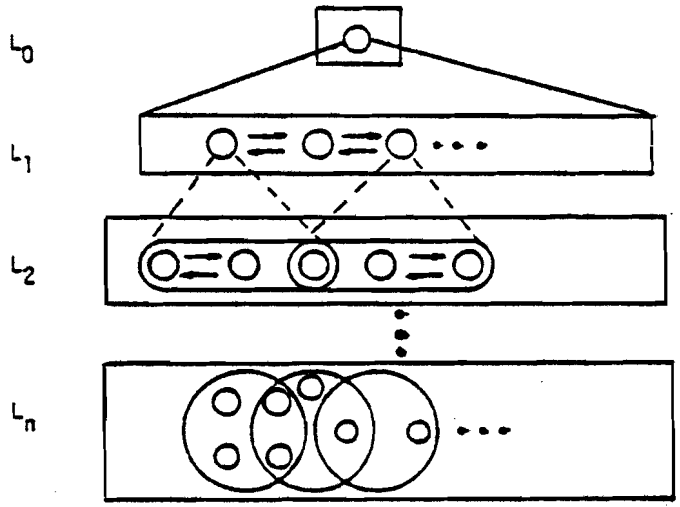

Figure 3. Hierarchical Design Approach

In these design approaches, the system is designed using a set of "subentities" whose behaviors and properties are specified and their detail is to be developed in a later stage. The specifications of these "subentities" should, therefore, include all the properties necessary for establishing the behavior and properties at the current level and only those properties necessary without restricting their construction at the later levels.

Another important concept has been developed recently in the area of system design is the family of systems approach. The essential concept is that when we are developing a system conceptually in a hierarchy of levels, we are starting out with a broad class of systems, and as we go along and make design decisions, the class of systems is limited to meet more specific requirements. If a system is designed with this philosophy, requirements changes and, in general, system modifications can be accommodated much more easily by zeroing in the proper level at which a design decision must be changed. A new branch on the system family is developed to meet the new or additional requirements.

The distinction we make between the software architecture design and the detailed software design is that software architecture design is more concerned with the logic of the problem and, among other things, is predominantly hardware independent; whereas detailed software design is concerned with the design decisions of realizing the system on a specific data processor. The major design decisions in software architecture design are: 1) identification of the functions the system must provide to respond to the environmental stimuli; 2) specification of the performance requirement of each system function; 3) identification of the major subsystems or subfunctions, information sets, and their interactions and coordinations; and 4) specification of requirement for reliability, maintainability, availability, etc. The major design decisions in software design, for comparison, are: 1) choosing the algorithms for implementing the system functions conforming to the functional and performance specifications; 2) choosing the major data structures (logical rather than implementational); and 3) designing operating system functions and scheduling of the functions (processes) on a specific computing system or systems.

The principles of hierarchy and abstractions apply also to the detailed software design level. Much of the effort of the software designer is occupied by the effort to coordinate the many activities in the system under stringent time constraints. The general problem is the correct and efficient coordination and synchronization of parallel processes. Most of the synchronization traditionally has been done by a general-purpose operating system. Because of efficiency considerations and the desire for decentralized control, more and more of these functions have to be performed at the application software level. System design methodologies involving parallel processes are being intensively investigated.

3.3.3. Software Implementation.

This phase is the coding step, normally done in high-level programming languages because of its demonstrated superiority over assembly languages. More and more efficient languages for the programmer are being developed, supporting structured code and data abstractions. The programming phase is actually beyond the realm of discussion in this paper; therefore, we are not going to explore further into its issues.

3.3.4. Operation and Maintenance.

System operation and maintenance refer to the phase of the life cycle after the initial release of the system. Requirements and the external environment of the system continuously change even after the system is operational. Furthermore, errors are found so that the original system must be changed. There has not been much attention paid to this phase of the life cycle; consequently, most of the system modifications are performed by ad hoc approaches. These have been characterized as design, implementation, and testing all over again. In principle, during the requirement definition and specification phases of the development stage, further modifications should be anticipated. In all the techniques surveyed, however, this aspect has not been addresses adequately.

3.4. Specification Techniques Classification and Evaluation.

This section prevents a framework within which the specification techniques that have been investigated can be categorized and evaluated. *The "technique" we refer to is generally a language or a scheme to express certain aspects of the system under development (e.g., specifications, requirements), and the associated tools, support, and logistics for analyzing them.* Occasionally, some techniques surveyed have a much wider scope. Under those circumstances, we extract their most significant features and discuss them along with others at the proper places.

We broadly classify all the techniques according to the form throughout the system development process with which it is best suited to be expressed. Thus we classify them as techniques for System Requirements, Data Processing Subsystem Requirements, Software Architecture Specifications, and Software Design Specifications. Within each category, subcategories are identified based on some of their characteristic features; they are then briefly described and evaluated as a group.

3.4.1. Specification Techniques for System Requirements (Overall Needs and Objectives).

As indicated in Section 3.3.1., system requirements, needs, and objectives are generally vague and ambigious, chiefly because they are at the top-level and arise directly from the application area problems. In practice, system requirements may also include the whole design context of the system, such as environmental design constraints and criteria for evaluating alternative designs. This information is stated in free form English and sometimes not explicitly stated at all.

Formalization is most difficult at this level. It requires complete codification of all knowledge relevant to the design of the system. However, we may hope to be able to formalize certain important aspects. Research work of special relevance here is in *semantic nets* and *fuzzy systems*. Semantic nets studies are mostly within the area of artificial intelligence. The main objective is to represent in the computer a body of knowledge so that computer retrieval and manipulation of this knowledge is possible. Information Automat is the only example initiated specifically for application in system requirements analysis. The method is still at a relatively early research stage. We expect it is best applied in small- to medium-scale system projects.

Fuzzy concepts and systems, pioneered by Zadeh, have now grown into a large body of knowledge. The chief objective of fuzzy concepts is to precisely formulate imprecise notions and relations that are characteristic of all large and complex systems. Although no specific work has been done on software system requirements analysis using this body of knowledgee, we do recognize that this is a new avenue worthwhile exploring. A design methodology suggested by Becker relies quite heavily on fuzzy concepts. (Becker discusses more of the philosophy than the methodology itself.) This seems to be useful in relatively small-scale system designs. The methodology is developed within the context of structural design in civil engineering. However, it is discussed in general design terminology so that its usefulness in software system design can be assessed without much difficulty and give a favorable impression.

3.4.2. Specification Techniques for Data Processing Subsystem Requirements.

The form and content, and hence the language, for data processing system requirements are the least agreed upon aspects compared to that of software architecture specification and software design specification. The requirements must serve two purposes. First, the user must be able to tell from the requirements whether he will accept the system and make other similar decisions such as evaluating certain specified properties of the system. Second, the system designer must be able to develop a set of software architecture specifications from the requirements and demonstrate that if a design is created satisfying the architecture specification, then the data processing system requirements are satisfied. *The basic requirement is that both the user and the system designer must be able to understand the document and perform meaningful evaluation.* We can narrow down the concepts and terminology in which the data processing system requirements can be stated. These must be in the overlap of the area of competence of the application-area experts and the system designer system concepts.

Several methods have been investigated and/or used, ranging from the unstructured form used to state the system objectives and requirements discussed in Section 3.4.1. to highly structured forms used to describe specific properties of the target system. The formal methods will now be discussed.

To tie together the numerous methods and techniques, the notion of a general model of a large-scale, real-time system (emphasizing the data processing functions) is developed.

The most general view of any system is to take it as a relation: $S \subset C \times Y$.

We restrict the class of systems to functional systems to simplify the notation (without loss of generality). Thus every system is characterized by a function

$S: X \rightarrow Y$

where X is the set of inputs and Y is the set of outputs. To model dynamic systems, we interpret X and Y as functions of real time, T, i.e., patterns of inputs and outputs (or input and output trajectories in the terminology of Wymore).

$X: T \rightarrow X$

$Y: T \rightarrow Y$

An example may make the abstract notation introduced so far more concrete.

Suppose system S has only one input X and one output, Y. Figure 4 shows a partial specification of S:

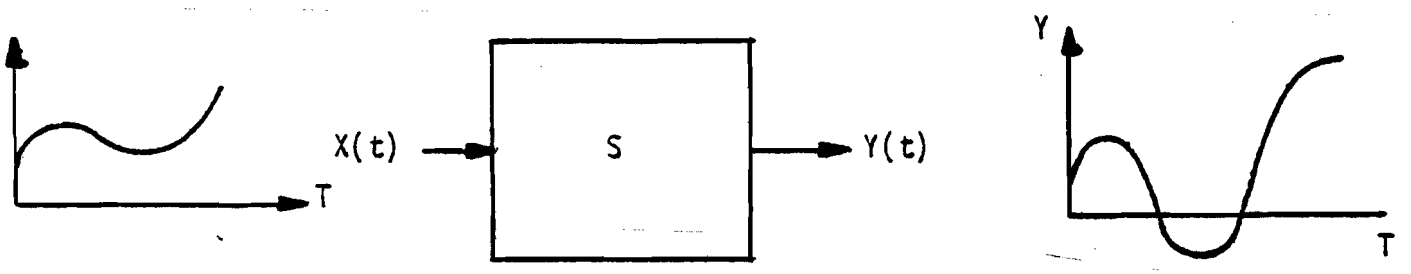

Figure 4: Functional Specification Example

A complete specification of S involves specifying all possible X(t) and the corresponding Y(t).

Performance requirements and other special attributes of the system can be specified by identifying the vector of performance parameters, W. The parameters may be implicit functions of X and Y. Thus the complete system requirements become:

S: $X \rightarrow Y \times W$.

There have been a number of studies conducted for methods of stating data processing system requirements (e.g., SREM, ISDOS). None of them is satisfactory for application in really large-scale, real-time systems development. We classify the requirements statement methods into three broad categories:

1) functional requirements; 2) performance requirements; and 3) specific attributes, with further subcategories wherever appropriate. Significant techniques in each subcategory are introduced and assessed. However, both the classification and assessment are somewhat subjective.

3.4.2.1. Functional Requirements.

Statements of functional requirements are specifications of the function S: $X \rightarrow Y$. Informally, it states the response the system should produce in reaction to a given stimulus. We identify three subclasses of methods.

3.4.2.1.1. Explicit Specification of S.

These methods usually employ rigorous mathematical statements. The most straightforward way is to define the domain and range of S, i.e., input and output spaces X and Y explicitly, and exhibit the function S by tabulation or a set of mathematical expressions. The method is, of course, limited by the ability of the user to state the function, S. In any real-word system, the complete specification of S is likely to be impossible. However, an explicit statement in such a rigorous form is most readily testable. Hence, formulating the critical functions of a system in this way to the extent possible may be of significant value for system level function validation.

Because of the difficulty in stating the complete function, another approach is taken by Fitzwater and Hamilton and Zeldin. The system function S is stated by a series of decompositions according to some decomposition rules. Thus S is composed of S_1, ..., S_n, and S_1 is composed of S_{11}, ..., S_{1m}, etc. The ability of the user to formulate his problem in the associated language is different, yet the rigor of precise mathematical statements permits certain properties of the function at various stages of decomposition to be checked. Apart from being less formal, the Structured Analysis and Design Technique (SADT) and a few others can be included in this category. These latter ones usually employ a graphical notation for better human communication with less emphasis on rigorous formalism.

There can be certain reservations in classifying the decomposition approach in the explicit specification category. The decomposition represents certain design decisions (non-arbitrary restrictions); the later design steps may be severely dictated by these early choices. We have made an assumption that design freedom should be maintained as far as possible. With regard to this, the function decomposition approach may as well be classified in the following category of implicit specifications of S.

3.4.2.1.2. Implicit Specification of S.

Methods in this category define the function S by exhibiting a procedure or structure to show how a certain output response of the system is generated in response to system stimuli and inputs. This can be done by a definition of the subsystem structures, functional partitions, etc. Generally speaking, it is more appropriate to regard this as the software architecture specification. A detailed discussion of this large category of methods is deferred to Section 3.4.3.

A comment as to the suitability for requirement statements, however, is appropriate here. Specifying a system structure as system functional requirements may be regarded as a limitation of design freedom. On the other hand, we may consider whether the limitation is unnecessarily constraining or not, and whether the nature of the application problem naturally suggests a particular system structure, and at what level of detail the system structure is specified as functional requirements.

A stronger argument for accepting them as appropriate specification techniques at this level is the recognition of a class of desgins as "wicked problems", and software system design is argued to be wicked. The term "wicked problem" was coined by Rittel, referring to design situations having the following characteristics (among others):

- Every formulation of the wicked problem corresponds to the formulation of the solution (and vice versa). The information needed to understand the problem is determined by one's idea or plan of a solution. In other words, whenever a wicked problem is formulated, there already must be a solution in mind.
- Wicked problems have no stopping rule. Any time a solution is formulated, it could be improved or worked on more. One can stop only because one has run out of resources, patience, etc.
- No wicked problem and no solution to it has a definitive test. In other words, any time a test is "successfully" passed, it is still possible that the solution will fail in some other respect.

If indeed we are dealing with wicked problems, specifying S (the problem) by exhibiting its structure (a solution) is the only means. Our position is that we should not consider "wicked" as a boolean attribute; different problems have different degrees of wickedness. It appears that in many cases, software system development is very wicked so that implicit specification of S is necessary in certain aspects but should be employed with careful considerations.

3.4.2.1.3. Optimization Model Formulation of S.

In many cases we do not have a functional requirement in the sense that given an input X, a specific Y should be produced. Instead, a large number of outputs would be acceptable, except that we prefer one to the others.

This type of situation is best formulated as an optimization problem. An example of system requirements specification of this form is:

- The set of alternatives, M, is specified (or to be investigated).
- The outcome function, C, specifying what would be the system behavior given an alternative
  $C: X \times M \rightarrow Y$.
- A valuation function, V, specifying how the combination of a certain choice and behavior (output) is valued
  $V: M \times Y \rightarrow R$.

Then the system functional requirements $S: X \rightarrow Y$ is specified by $S(x) = y = C(m, x)$ if $\forall m_i \in M$,
$V[m, C(x, m)] \leq V[m_i, C(x, m_i)]$.

Other variations of this formulation are possible.

The above example is in a very crude form because there is no investigation of this nature within the discipline of software engineering. Nonetheless, fruitful results may be obtained with some investigation. Another remark applicable here is that the performance requirements of the system are even more naturally formulated in this class of models (this will not be repeated in the following paragraphs).

2.4.2.2. Performance Requirements.

Performance Requirements need additional information for the complete specification of S: $X \rightarrow Y \times W$, where W is the vector of the performance parameters.

The most difficult part of the requirements statement is the definition of all the relevant performance parameters. There is essentially no notion of completeness as far as performance requirements are concerned. Functional requirements can be regarded as complete if S: $X \rightarrow Y$ is defined for every possible $x \in X$. However, W is essentially an open-ended list of parameters. For a chemical process plant control system, the pollution level can be a performance parameter. Leaving it out in the requirement specification does not constitute logical incompleteness unless it is already recognized that the factor is relevant.

In practice, there is currently no formal method of stating performance requirements except in free-form English descriptions. The most important performance requirements seem to have been timing (response time, deadline specification, etc.) and loading factor (the amount, frequency, and distribution of inputs and outputs).

Methodologies that include special provisions to state (partial) performance requirements are the System Requirements Engineering Methodology (SREM), ISDOS, ADS, etc. In SREM, the performance requirements are stated by specifying "validation points" in the R-nets and the performance information to be collected at these validation points. Performance requirements specification in other methods are more or less similar, including the specification of the frequency and distribution of certain data, the deadline for certain outputs, etc.

As an overall remark, performance requirements specification is another area that has not been addressed adequately in the past work.

3.4.2.3. Specification of Special System Attributes.

A typical requirement statement regarding system flexibility may be as follows: "System growth against evolving external load shall be achieved without major subsystem redesign, without major impact on other subsystem components, and without extensive system inoperability".

The above statement exhibits both the ambiguity ("major", "extensive") and untestability of requirements statements. Attributes such as flexibility, security, relability, etc., must be formulated precisely for meaningful analysis; otherwise, the decision as to whether a system has these attributes will be highly judgmental. Work in this direction of formalizing special system attributes is now emerging, mostly originated from operating systems research. Notions such as deadlock free, security, and integrity have been investigated. It is not yet certain whether this can be applied in a more general context.

3.4.3. Specification Techniques for Software Architecture.

Currently, there is not a generally accepted theory of what should be considered as the basic concepts and objects of a system and how they should be organized as a "structure". In any event, *the software architecture specifications should be an implicit definition of the system function S* as introduced in Section 3.4.2. Furthermore, a method (or language) for the specification of large-scale software architecture at the system level should reflect the way a system is conceived by the requirement analysts. In principle, the formal algorithm specification of both sequential and parallel processes, and data abstraction to be discussed in Section 3.4.4. may be applied at this leven also. However, the current state of development of these techniques is considered as impractical at the system level.

Other software architecture specification techniques center on two dominant features of a real-time system: the data flow and the control flow. In some systems, the flow of data is so dominant that having a clear picture of the flow and interrelation of the system elements almost completely tells us all about the system.

In others, with intricate synchronization and timing, the control aspects plays the more important role. Many systems, of course, consist of both aspects. Techniques for specifying the data flow type of systems are considerably more developed and numerous than the control flow type, although increasing efforts are being directed to the control flow type as real time, control, and planning systems are becoming more complex and critical. Discussions of the software architecture techniques that have been and are being investigated are divided into two categories dealing separately with the two types of systems.

We must, however, emphasize that the classification should be considered as loose, because many techniques address both features together. Features other than the above two (though not justifying separate categories) may also be included in some techniques.

3.4.3.1. Methods for Specifying Data Dominant Systems.

The typical data flow dominant or data driven systems are business oriented information processing systems. In this type of systems, a set of output documents is required to be produced at specific times or at regular intervals. The main concern of these systems is how these output documents are derived from the input data, and internally, what data are to be retained for later use, etc. Once the process of getting the output from the input is specified, the structure and the rest of the system become apparent. An example is a simple payroll system. The required output is a weekly paycheck to every employee, and the inputs are daily work records and new employee records. The data flow requirements are shown in Figure 5. The control aspect follows naturally from the data flow requirements.

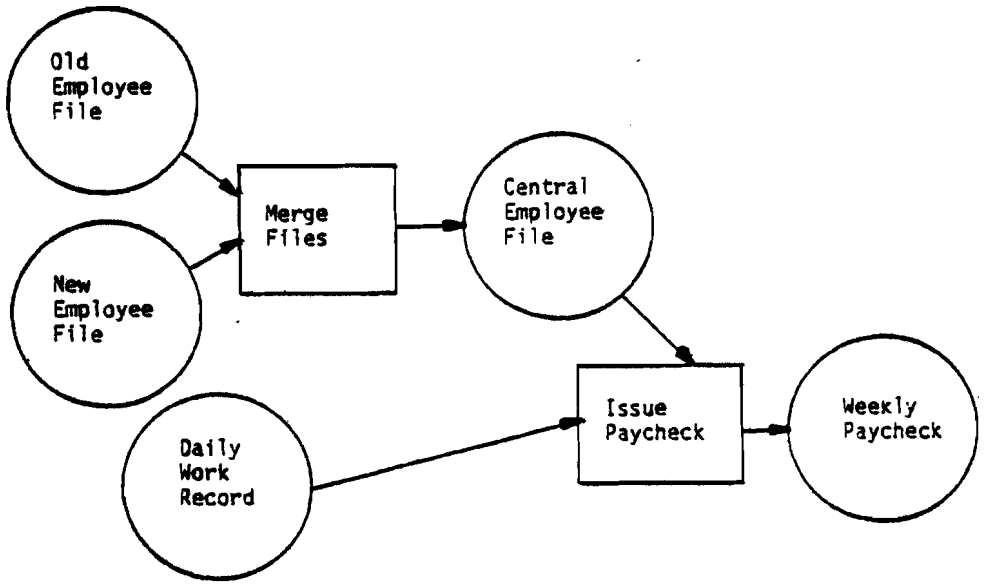

Figure 5. Simple Payroll System Data Flow

Most of the previous work in system specification deals with this class of systems. These approaches, however, have different motivations and objectives. A large portion of them are for documentation purposes only. These stem from the need for unambiguous communication among people involved in the system development and ease in systematically cross-referencing a large volume of information. There is little analysis performed on the specifications. In almost all cases, a language, a set of forms, or a discipline is developed.

The following provides an overview of their basic common features, rather than emphasizing their differences. References to a specific approach are made when its unique features are discussed. Special attention is placed on the techniques for analysis of the specifications.

A typical approach usually has provision for specifying one or more of three groups of information: 1) the data flow network, 2) process or function specifications, and 3) other information.

3.4.3.1.1. Data Flow Network.

The specification or requirement statement language provides the basic capabilities for stating what data or documents are required of the system, and the data requirements and processes for the derivation of each output. For example, in Figure 6, c is the required output and a specification statement may be as follows: Process P, with inputs a and b, is needed to generate c.

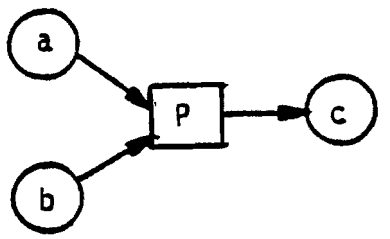

Figure 6. Example Data Flow Requirement

Some of the inputs to the process may be internal file data (usually referred to as history-type data). This procedure is continued until conceptually a network of data flow is constructed (Figure 7).

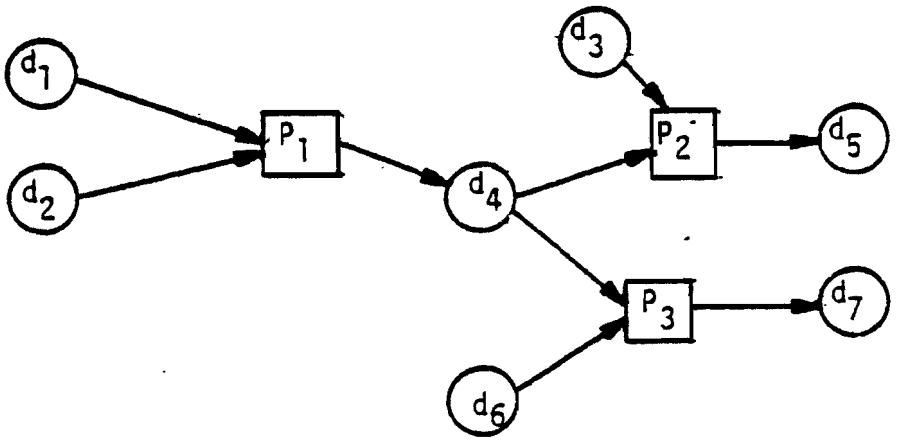

Figure 7. Example Data Flow Network

In Figure 7, the squares represent processes; and circles, data items. A graphical representation of their relationship is amenable to a number of basic analyses--typically connectivity and completeness (all data not feeding to some processes must be external output and all data with no source must be external inputs or history data). Other analyses that can be performed include identification of strongly interacting subsystems, restructuring, dynamics, etc. These are based on criteria that are less universally adopted.

3.4.3.1.2. Process or Function Specification.

To specify the process or function is to define the functional relationship between the inputs and outputs of the process. If these are simple data elements, the functional specifications are easy, especially in management information systems where the relationships are typically arithmetical.

More complicated situations arise when the inputs or outputs are aggregates of simple data elements (e.g., a file of records). For instance, a process has to calculate the total net pay of an employee from a file of pay rate and a file of work time.

There are attempts to develop a language that is capable of expressing these relationships among data (without specifying implementation). A formal approach is the information algebra developed by the CODASYL Development Committee.

Among the key concepts introduced are lines, areas, bundles, glumps, and functions of them. Lines correspond (roughly) to records and areas to files. A bundle or a glump specifies a new file based on data in other files. These descriptions given here are vague because the original notions are built up from a rather elaborate basis.

Some additional concepts are yet to be developed beyond simple lines and areas for more general applications. In the end, this can be extended to a comprehensive data base language. The possibility of using such a language for specification of data flow and manipulation should not be ignored.

3.4.3.1.3. Other Information.

Besides the data flow and processing function specification, some additional information can be specified within the existing framework. Among these, the most common ones are timing and system load information. Timing specification deals with requirements of the following types: 1) some output documents must be provided before a certain deadline, 2) certain outputs have to be produced periodically, and 3) certain outputs have to be produced after an input event within a specified response time. System load specifications include size of file, maximum, mean, minimum, frequency of usage of data, etc.

Analysis of this information may be important in some systems. Information such as a calendar of events or the volume of data flow among subsystems can be extracted from these specifications. More elaborate analyses are based on a graphical model of the static flow and volume information. We illustrate the approach with an example. Given a network of data flow as in Figure 7, let $P_1, \ldots, P_n$ be the set of processes and $d_1, \ldots, d_k$ be the set of data sets. An incidence matrix of the processes and data sets can be obtained, which is defined as follows:

$$e_{ij} \begin{cases} 1 \text{ if } d_j \text{ is an input to } P_i \\ -1 \text{ if } d_j \text{ is an output from } P_i \\ 0 \text{ if there is no direct incidence between } d_j \text{ and } P_i \end{cases}$$

Let v_j be the volume (e.g., storage size) of d_j; e_i be the number of inputs and outputs for P_i; and m_j be the number of times d_j is used either as input or output, then

$$\bar{e}_i = \sum_{j=1}^{k} |e_{ij}|, \quad i = 1, 2, \ldots, n$$

$$m_j = \sum_{i=1}^{n} |e_{ij}|, \quad j = 1, 2, \ldots, k$$

The transport volume for d_j is $t_j = m_j v_j$, and the transport volume for the whole network is

$$T = \sum_{j=1}^{i} m_j v_j$$

This may be taken as a measure of the amount of data movement necessary between the main and secondary storages.

3.4.3.2. Methods for Specifying Control Dominant Systems.

In some real-time systems, the control flow or the sequence of operations of the system becomes the dominant feature for consideration. Many important aspects of a system are characterized by its control flow. The operation sequence, the interaction pattern of the subsystems, the deadlock situation, etc., are among the more important ones.

In designing a system, it is desirable to have these features validated before commitment to further design and implementation. Therefore, specification of the control flow of a system is important even at the earliest stage of the design.

An approach in this category generally develops a graph model of the system abstracting the structural aspects, so that some formal properties of the system can be analytically examined, and desirable and undesirable features identified.

Here we will examine the theoretical bachground of this group of specification techniques without going into details of their specific setup.

Graph models have been developed to study various aspects of computer systems and other more general systems. Originally, the more important issues considered are assignment and sequencing of computations, harmonious cooperation of processes, memory allocation, transformation of sequential processes into parallel ones, etc. Currently, much interest has developed for extending these studies to model system level requirements of large-scale software.

The major advantages of using a graph model for the representation and analysis of system specifications and requirements are fourfold: 1) many graph models are capable of modeling very general features of large systems, especially the asynchronous interactions of parallel subsystems; 2) abstractions--a graph model retains certain properties and leaves out irrelevant detail (this makes them particularly attractive for formal analyses); 3) the existence of a variety of theoretical techniques for studying the models; and 4) visual convenience--a graph can be an excellent aid for visual understanding, which is very useful for human analysts.

To improve the concreteness of the discussion, we will use the Petri net as an example for sampling the available analysis techniques on the model. (Many of the theoretical considerations have a direct analog in other models.)

A Petri net is a bipartite directed graph, which means that it consists of two types or sets of nodes (usually represented by circles and bars) and a set of directed arcs going from a member of one set of nodes to a member in the other set. The two sets of nodes are places (circles) to be interpreted as conditions or events, and transitions (bars) to be interpreted as processes or actions. A dot (token) may or may not be present at each place. When there is a token in a particular place, the place is referred to as a condition being true or the condition holds. Figure 8 shows a Petri net. The places are c_1, ..., c_5 and the transitions are t_1, ..., t_4. Conditions c_2, c_3 are holding while the rest are not. The pattern of tokens in a net is called the marking. A net models the following situation: for each transistion, if every input place (places with a directed arc to the transition) is holding, then the transition is enabled and can fire. After it fires, the output conditions will hold, i.e., each of the output places receives a token. Thus a net represents all the potential asynchronous actions.

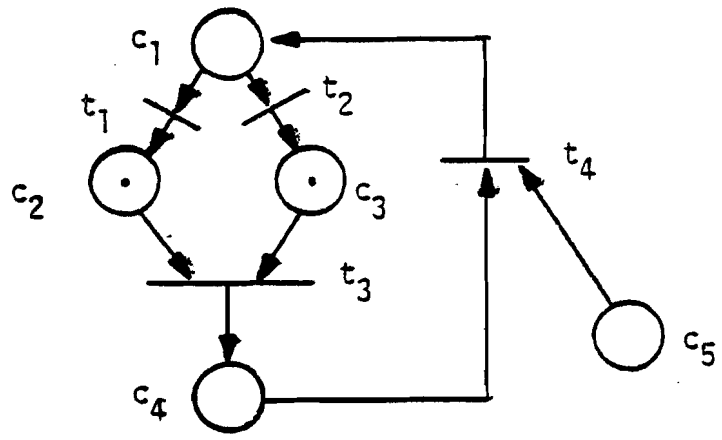

Figure 8. Petri Net Example

Many properties of a net can be studied. A net is live if all its transitions are live. A transition is live given a particular marking if there exists a sequence of firings that fire it for every marking reachable from the given marking. A marking is reachable from another marking if there exists a sequence of firings transforming the latter into the former. A live net roughly corresponds to the situation of the absence of deadlocks. A net is safe if all places are safe. A place if safe given a particular marking if every marking reachable from the given marking has at most one token on that place. Other theoretical problems pertaining to Petri nets and generalized Petri nets are the reachability problem and the liveness problems that address the problem of whether a particular marking can be reached and whether a given marking of a net is live, respectively. Many of these problems are difficult decidability problems.

3.4.3.3. Specification Analysis System.

The techniques already covered in this section are intended to be used over the extended period of software architecture design. For a large project, very voluminous information is generated among a large group of system designers. Automated aids are needed for managing this situation. In more recent research, it usually takes the form of a specification analysis system so that the system designers can input their specifications, which are deposited in a central data base. A collection of automated tools will be available to perform various analyses on the specifications. The most representative of this type of system is found in ISDOS and SREM. In the following we will briefly describe some of the technical features of a typical specification analysis system--a schematic diagram of which is shown in Figure 9.

3.4.3.3.1. Monitor.

The monitor is the interface between the user and the analysis system itself. Instrumental to the success of a specification system is the nature and extent of interaction provided by the system. For this reason, considerable attention must be given to the development of suitable feedback mechanisms and formats. An important design consideration of the monitor is that it should be able to guide the user through a complete specification analysis, while the system automatically maintains a partially entered of changing set of specifications.

Other supplementary functions performed by the monitor may include configuration and management control of the development project, and automatic generation of documentation in human readable form, etc.

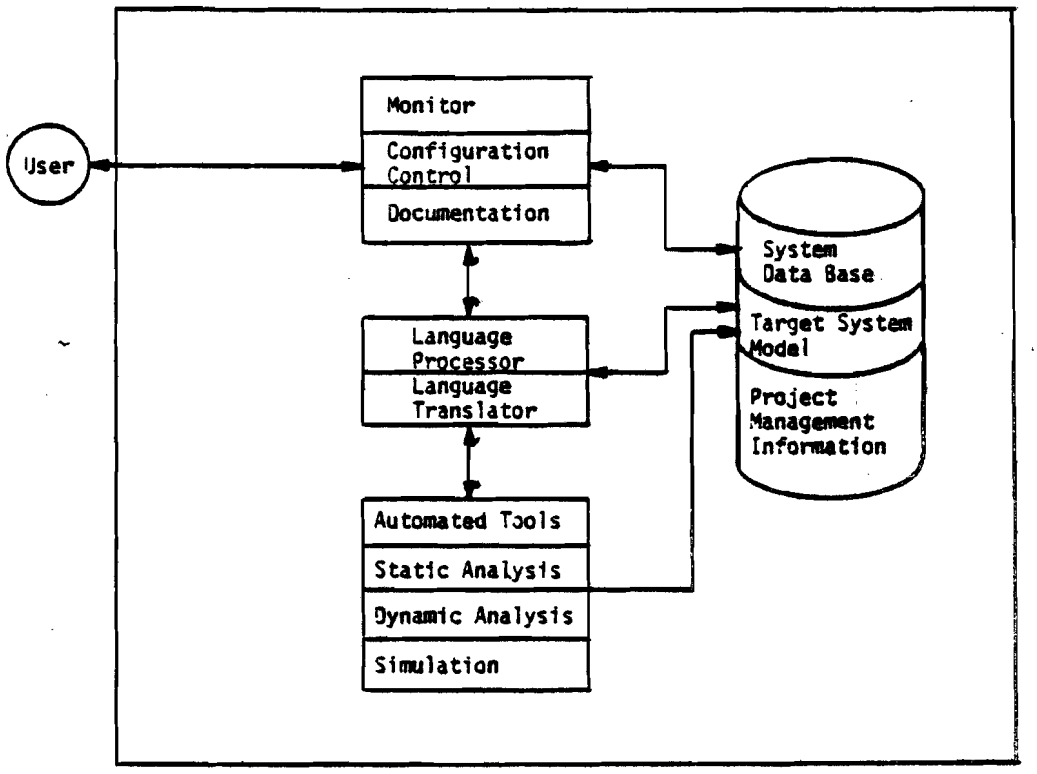

Figure 9. Specification Analysis System

3.4.3.3.2. Language Processor.

The language processor will perform the usual function of analyzing the input statements written in the specification language accepted by the system. Syntactic analysis and other diagnosis results are immediately feed back to the user. The accepted input is deposited to the central data base via the language translator.

3.4.3.3.3. System Data Base.

The most important part of the system data base should contain a computer representation or model of the target system under development. This model will be continuously modified under appropriate configuration controls, including addition, deletion, modification, and entry of alternative versions. The representation is directly submitted for analysis.

3.4.3.3.4. Automated Tools.

A collection of analysis tools of the system data base is usually the key component of the specification analysis system. These tools, however, are strongly dependent on the specific techniques used for specification and their underlying concepts. The general features include the static analysis tools that check out the necessary and desirable features of the data base such as consistency and completeness. A variety of dynamic analysis of the target system can be performed depending on the system model use.

The most sidely used analysis technique for studying the dynamic behavior and performance of a system at its specification stage is simulation. Simulation can be generated either automatically or manually. The specific analysis tools have been discussed along with the specification techniques in paragraphs 3.4.3.1. and 3.4.3.2.

3.4.4. Specification Techniques for Detailed Software Design.

Because the level of detail in specifying a detailed software design is much lower when compared to specifying the software architecture, the techniques in this section are applicable to higher levels if not limited by the complexity and by minute details. We will examine those techniques that try to describe the entities concerned down to the most primitive details, although they may be built up hierarchically by a number of levels.

There are, loosely speaking, two aspects for specification--a unit of action and a class of objects. A unit of action, in current high-level programming language terminology, may be a procedure, block, or subroutine. The overall system functions are ultimately built up of these basic units of acion, e.g., a sorting routine, a Kalman filter procedure, etc.

Specification of a unit of action, which we will later refer to as process specification, is to specify succinctly all the effects that the action will have on its environment. The primary objective is to satisfy the minimality criterion that all the relevant, and only that, information needed to use the unit is stated. In general, the implementation information is not relevant.
The second criterion is comprehensibility, i.e., the user should be able to understand all the effects explicitly. This also requires that the specification be unambiguous. The specification should also allow a unique interpretation. The two aspects are simultaneously satisfied only by a formal but natural notation.
The third criterion is whether the method allows formal manipulation, in particular, proving that an implementation satisfies the specification and that the specification itself satisfies some consistency requirements.
A unit of action may belong to one of the two classes requiring very different specification techniques because of a fundamental difference between them. These two classes are sequential process specification and parallel process specification. A sequential process can be thought of as one that will be in action alone (although some other processes may be active concurrently, they have no influence on one another). On the other hand, in specifying parallel processes, specific attention must be paid to their interactions with one another.

The idea of "data type" or data abstraction is to treat the objects in programming as abstract entities instead of their physical implementation. The objects are treated as primitives and their semantics are completely defined in terms of the set of primitive operations and relations on them. These are similar to the definition of mathematical objects, treated as undefined concepts, and constructed inductively by primitive constructors.
In the following, specific classes of techniques are outlined under the headings of methods for specifying sequential processes, parallel processes, and data types.

3.4.4.1. Methods for Specifying Sequential Processes.

3.4.4.1.1. Enumeration of Input/Output Pairs.

In this method, the function is stated as a list of input/output pairs. For example, to define the square function on the integers: f = ...(-1,1),(0,0),(1,1),(2,4).... Although this form is regarded as a specification, it is impractical in almost all situations and is theoretically impossible in some cases (e.g., for infinite input domain funcions).

3.4.4.1.2. Exhibiting a Procedure to Obtain the Output From the Input.

A procedural description may be used as a specification with emphasis on clarity of expression, whereas the actual implementation emphasizes the efficiency of the process so that a completely different algorithm may be used. The specification (possibly simpler in structure) is taken as "correct" or as "what is desired". The correctness of the actual implementation is validated against the specification.

In terms of the desirable criteria of a specification method, minimality is not satisfied since the specification of a procedure includes much information that is not needed. This is the major criticism of the procedural specification method. The user, say a programmer, may make use of some specific detail of the procedure, which may be changed in a new version. Substantial changes in a large portion of the system are necessary in this situation.

Furthermore, rigorous verification cannot begin until the whole program or system has been completed. This implies that the verification effort is complex and no corrective action can be taken until the very end.

With regard to comprehensibility, it depends on the programming style, structure, and more importantly, the size of the procedure. In general, this is not a critical problem because we are becoming aware of some principles of program clarity and readability.

3.4.4.1.3. Input/Output Assertion.

The actual procedure of how to derive the output from the input is not stated. Instead, the relation between the input and output variables is specified in terms of more primitive relations and notions.

In the approaches within this category, the actual implementation detail is left out. Only the entry and exit conditions are stated; the user only needs to ascertain that the input condition is satisfied on the invocation of the procedure, and it is guaranteed that the exit condition is satisfied on termination. Verification involves showing that the implementation has this property.

Because the conditions are stated in terms of more primitive notions, the criterion of minimality may not be satisfied in that the choice of the particular primitive notions to implement the algorithms is arbitrary and may be a specific one from among many possibilities.

The comprehensibility of this method again depends on the complexity of the procedure and also on the choice of the suitable level of the primitive notions used.

There are two difficulties to the approach: (1) The growth in the complexity of the condition when the system or procedure to be specified becomes complex. Even with the best design of a hierarchy of the primitive notions, the complexity may still get out of hand; (2) The "naturalness" of stating the entry and exit conditions. While it seems natural to express some function, for other functions it may not. A good example is a "differentiate" function, it is more natural to show how to do it than telling what it is all about. On the other hand, stating the entry and exit conditions is far simplier in the case of matrix inversion, etc.

3.4.4.2. Methods for Specifying Parallel Processes.

Specification techniques for parallel processes have not been studied as extensively as those for sequential processes. Their importance is now apparent since actual parallel hardware is gradually being employed. More importantly, a system viewed as a society of parallel processes is conceptually cleaner and more elegant. Principato has performed an extensive survey of these techniques; The techniques are classified into three categories.

3.4.4.2.1. Parallel Constructs in Programming Languages.

Program languages are extended with special primitives to indicate interactions among parallel processes. Thus, for example,

$$\underline{\text{cobegin}}\ P_1, \ldots, P_n\ \underline{\text{coend}}$$

indicates that the process P_1 through P_n will be executed concurrently. If any of these processes refers to some common data, the computation results are usually indeterminate.
For a specific application, these random interactions must be restrected. The most common example cited is the reader and writer problem. Two groups of processes, designated readers and writers, are working on a shared data base; the readers only refer to information in the data base, while a writer may modify it. To guarantee predictable results, whenever a writer is working on the data base, no other processes may be permitted to use it, even for reading only. However, any number of readers may use the data base simultaneously. This and other problems of synchronization and communication, and other criteria such as fair and deadlock free scheduling, etc., lead to the development of the semaphore, critical region and conditional critical region, and monitor primitives.
Using these primitives to specify parallel processes is analogous to the techniques described in paragraph 3.4.4.1.2. for the sequential case. They share the same comment that an actual solution is exhibited rather than stating just what is intended.

3.4.4.2.2. Event Approach.

Techniques in the last category integrate the sequential processing aspects with the parallel synchronization aspects. It is possible to isolate the latter so that they can be examined separately. A method using the event approach, such as Habermann's path expressions, tries to identify the allowable sequences of events from all the processes. For example, in the reader/writer problem described in the last paragraph, the sequences of events allowed are those in which an exit from the shared data by a writer i immediately follows an entry of the same process i.

These techniques have the merit of exhibiting the synchronization explicitly, so that their properties can be proved formally. Deadlocks and other undesirable situations can be uncovered.

3.4.4.2.3. State Variable Approach.

In the state variable approaches, variables are introduced sometimes additional to the normal program variables to specify the states of the system of the parallel processes. Constraints on the possible states that can be taken are stated as a set of invariants involving the state variables. These invariants are supposed to characterize the synchronization. A simple example is the synchronization of a producer and consumer sharing a buffer. A set of state variables may be the number of units produced, p, number of units consumed, c, and number of units in the buffer, b. Assuming that the buffer size is N, the constraints are:

$b \leq N$ and $c \leq p - b$.

Although formal proofs of desirable properties of the synchronization and the correctness of implementation can be obtained, it may be difficult to specify correctly the desired synchronization intended.

Specific techniques in this category are those of Robinson and Holt and Owicki.

3.4.4.3. Methods for Specifying Data Types.

There are a large number of researchers in this area.

An abstract data type is a collection of objects with a set of permissible operations specified. Any manipulations on these objects must be done via the permissible operations, and conversely, the characteristics of the objects are completely defined by these operations alone. The exact representation of the object is not part of the specification.

An illustrative example is the definition of a stack (Figure 10). The primitive notions are stack, item, boolean; and the primitive operations are MTSTACK (which creates an empty stack), PUSH, POP, TOP, and ISMTSTACK. The complete specification of the stack consists of six axioms (Figure 10).

declare	MTSTACK()→stack
	PUSH(stack, item)→stack
	POP(stack)→stack
	TOP(stack)→item
	ISMTSTACK(stack)→boolean;
for all	s ε stack, i ε item,
Axiom 1	ISMTSTACK(MTSTACK) = true
2	ISMTSTACK(PUSH(s,i)) = false
3	POP(MTSTACK) = MTSTACK
4	POP(PUSH(s,i)) = s
5	TOP(MTSTACK) = undefined
6	TOP(PUSH(s,i)) = i

Figure 10. Algebraic Axiom Specification of Stack

The most salient features of the stack are: POPing a stack results in the one just before the last PUSH operation, and examining the stack gives the last item pushed in.

These are captured minimally by the axiom system. It is minimal because it only involves those concepts that are relevant at the level of consideration of the stack.

The comprehensibility of this form of definition depends on whether the object being specified has operations with simple relations among them. It can be very complex, difficult to understand, and may involve more than it requires to state explicitly the algorithms of the individual operations themselves. Currently we know of no effective (general) method either to verify whether an implementation of a given data structure satisfies the specification or to verify automatically certain "correctness" or "consistency" properties of a set of axioms.

3.4.4.4. Assessment of Software Design Specification Techniques.

After this survey of the variety of techniques, a summary statement about the whole class can be made. So far we have been dealing with the specification of units of relatively small sizes. The information specified is complete and precise in the sense that everything needed is given. Because of this property, the complexity and comprehensibility become a stumbling block. The same problem of complexity exists even in a greater magnitude in the verification of the specification. Special techniques by relaxing the preciseness and completeness must be used in large systems. Some aspects of all the information is given up for concentration on more important features, namely, the structural aspects of the system.

Another major problem is in the construction of larger and larger units from basic blocks. We have seen roughly how this can be achieved in a hierarchical fashion in the discussion of algorithm specification. However, emphasis is not placed on the coordination and performance aspects of the subsystems of a system. All techniques discussed so far do not address these aspects. They become the prominent feature of a large system, and special techniques have to be developed for them, as have been discussed in Sections 3.4.1. through 3.4.3.

4. Design Issues.

4.1. Code Level Concepts.

4.1.1. Abstraction.

The concept of abstraction ranks as one of the most important advances in programming which has occurred in the last 20 years.

It is the basis for high level languages, virtual machines, virtual I/O devices, data abstractions, and a host of other concepts. The whole concept of bottom up design consists of building up layers of abstract machines which get more and more powerful until only one instruction is needed to solve the problem. In most cases people stop far short of defining that one super-powerful instruction, but they do significantly enhance the environment in which they have to program. Device drivers, operating system primitives, I/O routines and userdefined macros all are built on the concept of abstraction and all raise the level at which the programmer thinks and programs.

In many cases the objective is to abstract out many of complicated interactions which can occur when many users or user programs are sharing the same machine. In other cases a virtual machine is created to hide the idiosyncrasies of a particular machine from the user so that the resulting program will be more portable. When the Modular Programming era began in the 1960's many people had the hope that hundreds of reusable buildingblock programs could be abstracted and added to their programming library and that they could finallyu begin to "build on the work of others". Unfortunately, as Weinberg noted, program libraries are unique, everyone wants to put something on but no one wants to take anything out.

4.1.2. Communication.

A program communicates with both people and machines.

A program is meant to communicate its structure to the programmer as well as to give instructions to the machine. The life-cycle cost of operating a program in most cases depends far more on how well it communicates with people than on how much it was optimized.

4.1.3. Clarity.

It has been said that a person who writes English clearly can write a program clearly. In studying English we are first taught to read and then taught to write. In programming we are usually only taught to write. In fact one language which is famous for its "one liners" has been affectionately called a "write-only" language.

The structure of a good article, paper, or book is very important for clearly communicating ideas. *The structure of a program is equally important for communicating both the algorithm and the context of a problem solution.* A good program should readily reveal its structure to the reader.

The concept of "data hygiene" has been around for quite a while now. That is, you should leave data as you would like to find it. The concept of "program structure hygiene" has never quite caught on. Every new change seems likely to increase the unstructuredness of a program.

The "structuredness" of a program, of course, is not very well defined. There is still no generally accepted metric for measuring the goodness or badness of a program structure.

In the best of all worlds the criterion of clarity could be applied quantitatively. Lacking that, we'll have to stick with peer pressure applied in design reviews and code walkthroughs.

4.1.4. Control Flow Constructs.

The concept of limiting the number and type of control flow constructs is now pretty generally accepted.

4.2. Module Level Concepts.

4.2.1. Cohesion.

Cohesion is the "glue" that holds a module together. It can also be thought of as *the association between the component elements of a module*. Generally one wants the highest level of cohesion possible. While no quantititive measure of cohesion exists, a qualitative set of levels of cohesion has been suggested by Constantine and proposed in modified form by Myers. The levels proposed by Constantine are the following.

Coincidental cohesion is Constantine's lowest level of cohesion. In this case the component parts of a module are just there by coincidence. There is no meaningful relationship among them.

Logical cohesion is present when a module performs one of a set of logically related functions. An example might be a module composed of ten different types of print routines. The routines do not work together or pass work to each other but logically perform the same function of printing.

Temporal cohesion is present when a module performs a set of functions that are related in time. An INITIALIZATION module performs a set of operations at the beginning of a program. The only connection between these operations is that they are all performed at essentially the same point in time.

Procedural cohesion occurs when a module consists of functions that are related to the procedural processes in a program.

Communicational cohesion results when functions which operate on common data are grouped together. A data abstraction is a good example of a module with communicational strength.

Sequential cohesion often results when a module represents a partition of a data flow diagram. Typically the modules so formed accept data from one module, modify or transform it and then pass it on to another module.

Functional cohesion results when every function within the module contributes directly to performing one single function. The module often transforms a single input into a single output. An example often used is SQUARE ROOT. This is the highest level of cohesion in the hierarchy and as such is desirable whenever it can be achieved.

A program of any reasonable size will usually contain modules of several different levels of cohesion. In fact many modules exhibit characteristics of a multiplicity of levels simultaneously. Where possible, functional, sequential, and communicational strength modules should be given preference over modules with lower levels of cohesion.

While levels of cohesion can be useful guides in evaluating the structure of a program, they don't provide a clear cut methodology for attaining high levels of cohesion. Also, levels of cohesion do not allow us to say that program A is right and program B is wrong. They do, however, represent a definite step forward. Before levels of cohesion were introduced there was no recognized basis for compariason. Now one can say in some inexact way that structure A is probably better than structure B.

4.2.2. Coupling.

Coupling is a measure of the strength of interconnection between modules.

High coupling among program modules results when a problem is decomposed in a relatively arbitrary way. Often, this method of chopping up a large program complicates the total job because of the resultant tight coupling between the pieces. This latter type of decomposing has been called "mosaic" modularity.

The other extreme in structuring a program is to consider only pure tree structures. These structures give rise to the concept of hierarchical modularity and provide many advantages for abstraction, testing and subsequent modification. Jackson would accuse you of "Arboricide" (the killing of trees) whenever you deviate from a pure hierarchical tree structure.

Brooks said that: "I am persuaded that top down design (incorporating hierarchy, modularity and stepwise refinement) is the most important new programming formalization of the decade."

Dijkstra said: "The sooner we learn to limit ourselves to hierarchical program constructs the faster we will progress".

Modular programs can be characterized as:

1) Implementing a single independent function.
2) Performing a single logical task.
3) Having a single entry and exit point.
4) Being separately testable.
5) Satisfying a number of other rules which have been listed at length in the literature.

When these rules are followed, the result is a set of nested modules which can be connected together in a hierarchy to form very large programs.

When modularity is used without hierarchy, one is only able to implement independent functions which can be executed in sequence. This approach tends to work on small programs but can seldom be applied to complex programs without seriously compromising module independence, connectivity and testability. Only when the concepts of modular programming are combined with the concepts of hierarchical program structure can one preserve the capability for both implementing arbitrarily complex functions and maintaining module integrity.

Modularity can be applied without hierarchy in cases which lend themselves very naturally to the efficient use of a very high level language. Very high level language statements are examples of functions which can be implemented relatively independent of each other but can still be strung together sequentially in a useful form. Unfortunately for most applications, the design of a convenient, efficient, very high level language is very difficult.

Hierarchical modularity forms an extremely atractive foundation. for most of the other Software Engineering Techniques. While some of these techniques can be used without having a hierarchical program structure, the primary benefit can only be gained when the techniques are all used as a unit and build upon each other. Specifically, a hierarchical modular program structure enhances top-down development, programming teams, modular programming design walkthroughs, and other techniques which deal with improving the development process.

4.2.3. Complexity.

The control of program complexity is perhaps the underlying objective of most of the Software Engineering Techniques. The concept of divide and conquer as an answer to complexity is very important provided it is done correctly. When a program can be divided into two independent parts, complexity is reduced dramatically as shown in Figure 11.

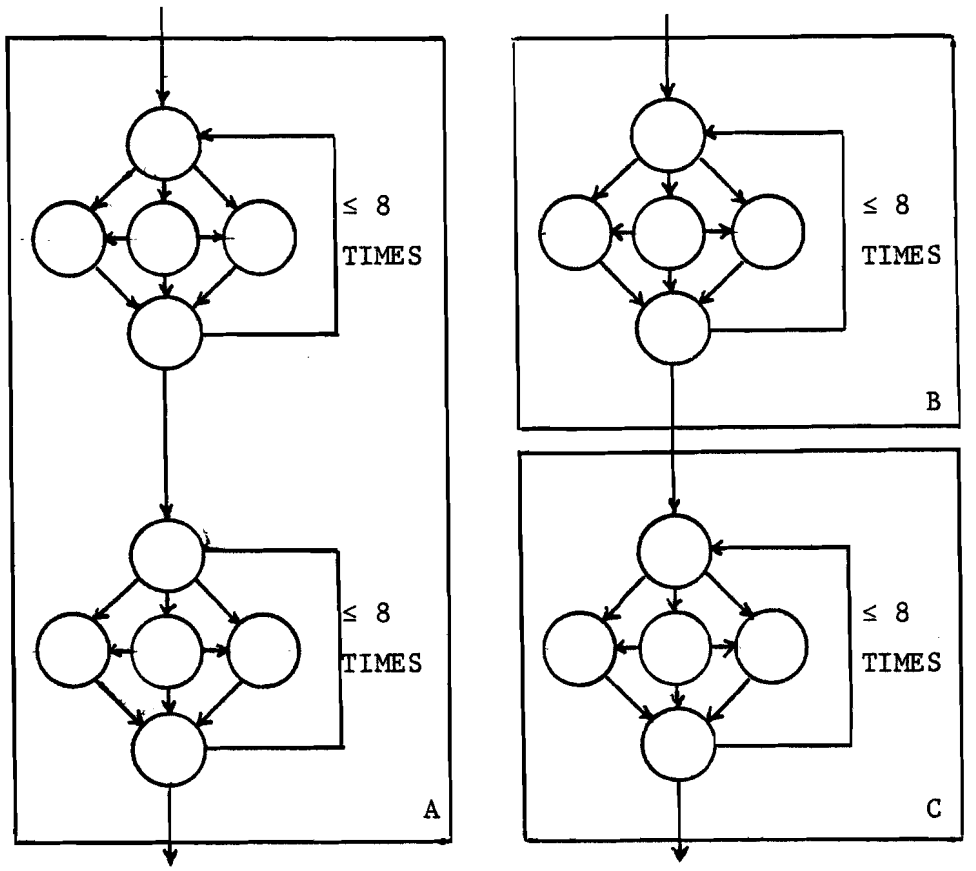

Figure 11

Consider program A where you have access to only the input and the output. A noble goal would be to "completely" test this program by executing each unique path. In the example shown there are approximately ~~250~~ 4000 billion unique paths through this module. If you were capable of performing one test each millisecond, it would take you ~~8~~ 130 years to completely test all of the unique paths. If, however, you had knowledge of what was inside the program, and recognized that it could be partitioned into two independent modules B and C which have very low connectivity and coupling, your testing job could be reduced. To test both of these modules separately requires that you only test the ~~one~~ two million unique paths through each module. At one millisecond per test, these tests would take a total of only ~~17~~ 33 minutes.

In this particular example from a testing viewpoint it is clearly worth trying to partition the problem so that small independently testable modules can be dealt with instead of just the input and output of a large program.
It should be clear from this example that the testing problem is best solved during the design stage. It is impossible to exhaustively test any program of meaningful size. Testing is experimental evidence. It does not verify correctness. It raises your confidence and may in fact increase reliability, but not very much.

4.2.4. Correctness.

A "correct program" is one which accurately implements the specification. In many cases a "correct program" is of limited value since the "specifications are in error". In many cases the specification is either excessive, incomplete, of inconsistent.
As discussed previously, correctness cannot be verified by testing. Its like searching for mermaids. Just because you haven't seen one doesn't mean they don't exist.
It is also unfortunate that for most problems mathematical proofs of correctness are as difficult to produce as a correct program. They have the additional disadvantage that a program must be reproven each time it is modified. You must write and prove your programs simultaneously.
Therefore the most promising approach for the near future may lie in finding a constructive proof of correctness. We will really have something if a design methodology can be found which leads one through the design process step by step and guarantees the correctness of the final program if each of the steps has been done correctly. While I don't remember that 321 × 25 is 8025, I do remember that 5 × 1 = 5,5 × 2 = 10,5 × 3 = 15, etc. Knowing these values and the steps of multiplication, I can rest assured that 8025 is indeed the correct answer. If only a design process existed which was as foolproof and easy to apply.
Without such a process we must live with a limitless capacity for producing errors.

4.2.5. Correspondence.

In Jackson's view, perhaps *the most crusial factor in determining the life-cycle cost of a program is the degree to which it faithfully models the problem environment*. That is, the degree to which the program model corresponds to the real world. All too often we are faced with a situation in which a small local change in the problem environment results in a large diffuse change in the program which purports to model that environment.

The world is always bigger than the program specification says it is, but the specification can always be extended if it corresponds to reality.

Since users tend to be gradualists, the changes in a realistic problem model will tend to be gradual. If the program is built around the static versus the dynamic properties of the problem, it can prove to be resilient to changes which occur over many years. While the program's model of the world cannot be complete, it must as a minimum be useful and true. If these criteria are met, many future maintenance and feature enhancement problems are avoided.

4.3. System Level Concepts.

4.3.1. Consistency.

One important objective of a good design methodology should be that of producing a consistent program structure independent of who is applying it. Given the same problem environment, three different programs which model that environment using the same design methodology should essentially be the same. Unless consistent designs can be achieved by different people, there can never be a true right or a wrong structure for a given problem.

One of the problems with design methodologies like Stepwise Refinement is that there is no one preferred, consistently obtainable solution. Instead, each designer seems to pull his own unique solution out of the air. There can never be a discussion of it being right or wrong, there can only be a discussion of my style versus your style.

4.3.2. Connectivity.

The harmful effects of high connectivity on system modifiablility can best be illustrated aby using an analogy.

Consider a system composed of 100 light bulbs. Each light in the system can either be on or off. Connections are made between light bulbs such that if the light is on, it has a 50-percent chance of going off in the next second; if the light bulb is off, it has a 50-percent chance of going on in the next second provided one of the lights it is connected to is on. If none of the lights connected to it is on, the light stays off. Sooner or later this system of light bulbs will reach an equilibrium state in which all of the lights go off and stay off.

The average length of time required for this system to reach equilibrium is solely a function of the interconnection pattern of the lights. The most trivial interconnection pattern is one in which all of the lights operate independently. None of them is connected to any of its neighbors. In this case, the average time for the system to reach equilibrium is approximately the time required for any given lamp to go off. The time for this to occur is approximately 2 seconds, and thus, the system can be expected to reach equilibrium in a matter of seconds.

At the other extreme, consider a case in which each light is fully connected to all of the other lights. In this case the length of time required for the system to reach equilibrium is 10^{22} years. It is clear that this is a very long time when you consider that the current age of the universe is only 10^{10} years.

Now consider one final interconnection pattern in which the set of 100 lights is partitioned into ten sets of ten lights each, with no connections between the sets, but full interconnection within each set. In this case, the time required for the complete system of lights to reach equilibrium is approximately 17 minutes. This dramatically shows the effect of connectivity. In terms of the concepts described earlier, this example represents very high cohesion within each module and very low coupling between modules.

Much as proper physical partitioning can dramatically reduce the time required for the system of lights to reach equilibrium, proper functional partitioning can dramitically reduce the time required for a program which must be changed to reach stability.

4.3.3. Concurrency.

Often, the proper model of a system will include concurrent asynchronous processes. Thus a program which properly models the system should consist of concurrent, asynchronous programs. There is often a producer-consumer relationship between processes. For example, process A could produce data which process B consumes.

4.3.4. Continuity/Change/Chaos.

As noted by Belady and Lehman a large program often appears to live a life of its own, idependent of the noble intensions of those trying to control it. Two important observations are summarized in the Law of Continuing Change and the Law of Increasing Entropy:

Law of Continuing Change: A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

Law of Increasing Unstructuredness: The entropy (disorder) of a system increases with time unless specific work is executed to maintain or reduce it.

These laws dramatize the key role played by the program structure during the life-cycle of software systems. The natural order of things is to produce disorder. If the program structure is unclear from the beginning, things will only get worse later. It is these two laws coupled with a poor program structure that have produced the maintenance cost horror stories.

4.3.5. Costs.

Michael Slavin has proposed that there is a fundamental truth of programmer arithmetic coming into play. That is, if it takes one programmer one year to do a job it will take two programmers two years to do the same job.

The magnitude of the problem is hitting people right where it hurts: in the pocketbook. Between $15 and $25 billion is now spent annually in the United States on software. People are gaining a life-cycle awareness which says that:

1) Software is a much bigger problem than hardware.
2) Maintenance costs often greatly exceed the cost of the initial development.
3) Design and Analysis is a much more difficult problem than coding.

This says that a good design methodology should catch errors early in the development cycle. The relative cost of fixing an error increases dramatically as one gets further along. The cost of fixing an error in the field can be two orders of magnitude greater than the cost of fixing the error during a requirements review.

4.3.6. Optimization/Partitioning.

All too often people confuse partitioning and design. Design is the process of deviding a problem and its solution into meaningful pieces. *Optimization and partitioning consist of clustering pieces of a problem solution into computer load modules which run within system space and time requirements without unduly compromising the integrety of the original design.*

There are at least three different types of modules which must be considered in programming. They are functional modules, data modules, and physical modules. Partitioning is concerned with placing functional modules and data modules into physical modules. In partitioning a program, several of these pieces of the program may be put together as one load module or even written together as one program.

It is in the packaging phase of a design where optimization should be considered for the first time. This phase is done at the very end and great care should be taken to preserve the program structure which you have worked so hard to create. In Jackson's words it is very easy to make a program that is right, faster. It is very difficult to make a program that is fast, right. Once an optimization has been cast in code it is like concrete. It's very difficult to undo.

5. Software Design Strategies.

5.1. Introduction.

Software design strategies exist today which can reduce the problems of software analysis, development, and maintenance and lead to less expensive, more reliable programs. Unfortunately, for many people these benefits have remained either nebulous or elusive. An attempt is made to explain and demonstrate the best techniques available and, where possible compare the results of using different design strategies on the same problem. We hope that this approach will make you aware of the new software techniques, such as it is, and may even help you to take a short but significant step toward writing correct, maintainable programs.

One format for discussing recent advances in the field of software engineering is to consider their effect on each phase of the software development cycle. That is, one could consider the effect of program teams and walkthroughs on problem specification. The effect of data abstractions on design, the effect of high-level languages on implementation, etc.

Here these advances have been divided into three different types of categories (see Figure 12).

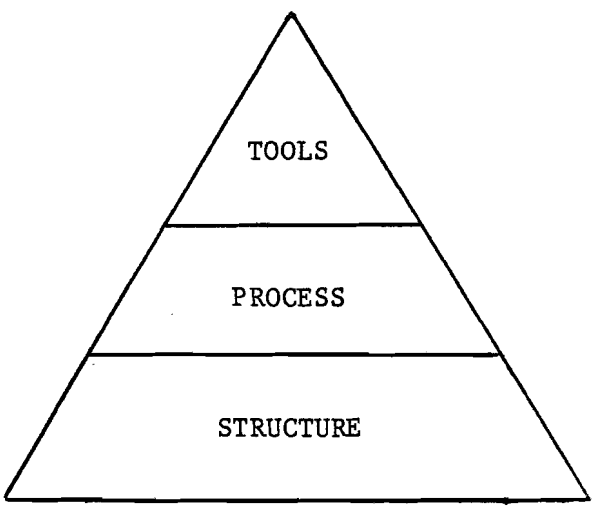

Figure 12.

The first category has to do with programming techniques and design strategies which directly affect the structure of the final program. Deriving a well-structured program which is an accurate model of the problem being solved and its environment is the single most important goal of the software design process. If the program has a structure with many random interfaces which do not properly model reality, no amount of effort or money expended in walkthroughs, high-level languages, or development support systems can keep that program out of trouble. It is only within the context of a well-structured program that walkthroughs, code reading, programmer teams, and development support libraries can be properly exploited. *Given strategies for designing well-structured programs, then one's attention can turn to improving the program development process.* A number of techniques which are organizational or management oriented are described which make sense if the design is done properly. Without a proper design, these techniques run the risk of becoming inefficient, ineffective, and expensive to administer. *Once methods for achieving proper designs have been found, and a process for implementing these designs has been achieved, only then should these procedures be automated and supported by development support tools.* Too often the tools are the first thing developed, without a proper understanding of what design strategies and development processes they are to support. This is wrong. The tools should support and enforce the design and development process after a conscious decision is made as to what that process is. If the tools come first, too often the design and development methods end up accommodating the tools instead of vice versa. The power of tools in determining what and how things get done was recognized early by Eli Whitney: "I will form the tools such that the tools themselves shall fashion the work." The approach taken here is to emphasize proper design strategies first and then note the requirements which they place on your development support environment.

Unfortunately, even though "structured programming" has been with us for more than a decade, we are still far from having all of the answers or for that matter even all of the questions. While it is clear that progress has been made, there is still much to be done.

5.2. Software Structuring Concepts.

Software design strategies which determine the structure of a program form the foundation for choosing and applying all of the other software engineering techniques. The development process can also be structured to exploit a clear and modular design. Development support tools can allow one to concentrate on real structural design issues instead of the more peripheral issues of scheduling, documentation and so on. In this section, some of the software engineering techniques which have their greatest impact on program structure will be introduced.

5.2.1. Modular Programming.

When modular programming became fashionable in the 1960's, it was characterized as: "Construction of a complete software system from a number of small functional units where there is a formal set of standards which control the characteristics of those units."
That formal set of standards usually included requirements like:

1) Modules implement a single independent function.
2) Each module performs a single logical task.
3) Modules have a single entry and a single exit point.
4) Modules are separately testable.
5) Modular programs are entirely constructed of modules.

In many cases the objective was to define a set of powerful, reusable modules which had the freedom and flexibility of a very high level programming language. The hope was to build on the work of others. Unfortunately, in most cases programmers were all too happy to share their modules with others but seldom sought out other people's programs to use themselves.

5.2.2. Structured Coding.

The next phase of structured programming started with concerns at a more microscopic level. That is, it addressed questions like "Is the code within a module easier to read, write, and maintain when it is constructed out of a limited number of control constructs which do not permit wild gotos." It was shown that the three basic constructs of sequence, iteration, and selection were sufficient to impelement the most complex programs and in fact formed the basis for writing more understandable, correct programs.
The major structured coding controversy occurred in 1972 and thereafter, partly in response to Dijkstra's widely acclaimed Turing Lecture. While much of this controversy, in retrospect, was blown completely out of proportion, it did serve to publicize the fact that the "software problem" might be amenable to change after all.

5.2.3. Hierarchical Modular Programming.

The concept of hierarchical modular programming gained a following in the early 1970's since it was viewed as carrying the advantages of structured coding above the module level. A hierarchical modular program structure is perhaps ideal in the sense that the connectivity of program modules is clearly limited and delineated. Subsequent inspection of modification of the program can be made in a straightforward manner since it is quite clear what parts of the program must be altered and affected. When diagrammed, the structure of such a program resembles an inverted tree structure. A faithfully implemented design will preserve this tree structure and with it obtain many advantages which will ease subsequent modification. An implementation which commits "arboricide" (the killing of trees) results in a disordered pile of leaves.

The modular part of hierarchical modular programming is applied in much the same manner discussed earlier. That is, one is still interested in implementing single endependent functions, performing single logical tasks, etc., but all of these constraints are applied to subtrees instead of single boxes on the structure diagram. The chief difference lies in defining the problem and its solution as nested modules which can become arbitrary complex rather than as an interconnected sequence of modules. While this approach limits one's options, it promises great rewards in easing later modification.

Note that many of the advantages of hierarchical modular program structure are topological. The connectivity of the program is severely restricted. Each module can be designed, tested, and modified independently. All connections with the rest of the system are clearly delineated and can be accounted for. An additional benefit is that the ripple effect of errors caused by program modifications can be dramatically reduced.

Note also that a final hierarchical modular program structure seldom reveals how it was derived. The following discussions focus on strategies for deriving program structures.

5.2.4. Bottom-up Design.

Bottom-up Design is the process of enhancing the capabilities of your machine by successively giving it more and more powerful instructions which hide more and more of the detail. In effect one builds up layers of virtual machines by forming data abstractions, resource abstractions, or high-level functional calls.

A popular example is the definition of a stack via data abstraction. The stack can be dealt with only through specially defined procedure calls (create, push,, top, etc.). These procedure calls "know about" the location and format of the data stored within, but hide that information from user programs. If the physical implementation or layout of data within the stack is subsequently changed, only the procedure-call programs must be modified. Since the user programs know nothing about the physical implementation of the stack, they are not affected.

Other physical input/output or storage devices can also be viewed as abstract resources. An example is the typical layered configuration of virtual machines in which an operating system kernel and operating system utilities insulate the user programs form dealing with the peculiarities of the hardware. Terminals are typically interfaced with user programs through "driver" programs. The concept of a virtual terminal which presents a uniform interface to application programs is an appealing one, since it can dramatically reduce the number of special cases the application program must deal with. Clearly, an application programmer wants to write information to a terminal without knowing or caring whether it is displayed on a CRT screen or printed or paper. Thus data abstractions are simply one example of abstract resources which are developed within the virtual machine seen by the application programmer.

The goal in most cases is to carry bottom-up design as far as possible while keeping a high degree of application independence. That is, the application programmer should build up a friendly virtual machine environment before solving any one application program in detail. If the virtual machine is designed properly, it can later facilitate moving (or porting) programs from one piece of hardware to another. Hopefully, the same virtual machine environment can be created on a number of physical machines, making the applications programs quite portable.

The whole concept of bottom-up design is very useful and powerful as long as one keeps in mind the fact that each program is modeling some aspect of the real world. One program should generally model only one type of terminal and this model should be detailed enough to support the functions it will later be called upon to perform. A program which models two different kinds of terminals will have unnecessary cross coupling, control interactions, etc.

5.2.5. Top-down Design.

Strong proponents of top-down design will encourage you to start a design by simply defining one super instruction which will solve the whole problem and then implementing that instruction with less and less abstract semi-super instructions. Sooner or later one gets down to instructions which will actually execute on the physical machine. This procedure is usually called functional decomposition, meaning that the main function is decomposed into successively simpler and simpler functions. Alternatively one can think of performing stepwise refinement, meaning that the solution is successively refined into more and more detailed explanations of how that solution is to be brought about. In either case, a parallel objective is to identify reusable functional modules wherever possible. In an ideal world, a very high level application-language or application oriented virtual machine would be defined along the way.

While this is a very noble objective, in practice it is extremely difficult to identify reusable modules in a top-down fashion, and what usually happens is that a fair amount of bottom-up design is done parallel. This gives the top-down designer the tools needed to solve the problems and provides a certain separation of concerns. Implementing the virtual machine and solving the application problem should be addresses as separate concerns even though they often have to be solved concurrently.

In doing top-down design, two questions must be confronted immediately:

1) What criterion do I use for identifying the top?

2) What is the basis for decomposing the problem and its solution?

Three strategies which have evolved for answering these questions in a reasonable way are discussed below.

5.2.5.1. Functional Decomposition.

In using this approach, the top is defined as the ultimate function to be performed and this function is divided into sub-functions by decomposing with respect to time order, data flow, logical groupings, access to a common resource, control flow, or some other criterion. The choice of "what to decompose with respect to" has a major impact on the "goodness" of the resulting program and is often the subject of much controversy.
The major advantage of functional decomposition is its general applicability. The disadvantages are its unpredicatability and variability (inconsistency, imprecision). The chances of two programmers independently solving a given problem in the same way are practically nil.

5.2.5.2. Data-flow Method.

The data-flow design method is essentially functional decomposition with respect to data flow. Each block of the structure diagram is viewed as a small "black box" which transforms an input data stream into an output data stream. When these are linked together appropriately, the computational process can be modeled and implemented much like an assembly line merging streams of input parts and outputing streams of final products. The only problem with this decomposition is that it tends to produce a network of programs and not a hierarchy of programs. This shortcoming is solved by Yourdon and Constantine by simply picking this network up in the middle and letting the input and output data streams "hang down" from the middle. At each level a module can be further decomposed into a "get" module, a "transform" module, and a "put" module.

Where as in the data-flow context the relation between modules was motivated by a "consumes/produces" relationship, this transformation procedure results in modules which are related by a "calls/is called by" relationship. Thus the hierarchy formed is really being artificially imposed by the scheduling and has nothing to do with modeling the problem in a hierarchical fashion. This contrasts with the "uses" relationshiop of decomposition with respect to functions, and the "is composed of" relationship which motivates a data structure design.
We feel that the system flow diagram which forms the basis for decomposing with respect to data flow is a very useful contribution and in fact may be the best approach currently available at the system design level. We are not convinced that the second step of putting things into a "calls" hierarchy using "transform centered design" is as useful. In fact it seems to produce a structure with a log of data passing and artificial "afferent" (input) and "efferent" (output) ears while reverting back to standard functional decomposition for decomposing the "central transform" which is the heart of the problem. It isn't clear that anything is gained over simply using functional decomposition from the start.

5.2.5.3. Data-structured Method.

When the program structure is derived from the data structure, the relationship between different levels of the hierarchy tends to be a "is composed of" relationship. That is, an output record is composed of a header followed by a report body, followed by a report summary. This relationship is generally a static relationship which does not change during the execution of the program and thus forms a firm base for modeling the problem.
Since a data-structure specification usually tends itself well to being viewed as correct or incorrect, the program structure based on data-structure specification can likewise be viewed to be correct or incorrect. This means that two people solving the same problem should come up with essentially identical program structures.

Michael Jackson, the inventor of the data structure design method, claims that it encourages a proper modeling of the problem environment and provides for a separation of concerns during the problem solution. Specifically, the programming process can be partitioned into the following steps:

1) Specify the virtual machine.
2) Define the data-stream structures.
3) Derive and verify the program structures.
4) Derive and allocate the elementary operations.
5) Write the structured text.

These steps can usually be performed and verified independently, effectively partitioning the design process as well as the problem solution. The resulting program structure for large problems is a network of hierarchies. That is, each simple program is implemented as a true hierarchical modular structure, but these simple programs are connected together in a data-flow network. The network can be placed into a "calls" or "is called by" hierarchy by a process called "program inversion", but this is considered to be a matter of scheduling which is independent of the structure of the program.

Both Jackson's and Warnier's data structure design methods were applied first in the area of business data processing. At this point they have also been applied to a number of on-line problems but are still unproven for most real-time applications.

5.2.5.4. A Programming Calculus.

While a "proof of correctness" is disappointingly difficult to develop after a program has been written, the constructive proof-of-correctness discipline taught by Dijkstra and Gries is encouraging. Dijkstra's design descipline can be methodically applied to obtain a modest-sized "elegant" program with "a deep logical beauty". Using this method, the program is produced as a by-product of the correctness proof instead of vice versa.

The initial design task consists of formally representing the required result as an assertion stated in the predicate calculus. This result assertion is then weakened to form an invariant relation which must hold true throughout the computation. The method is a top-down method to the extent that both the result and invariant should be formed in stages by a sequence of stepwise refinements, but the ultimate result is a host of logical assertions which must be specified formally, using the predicate calculus.

Many people call it a disadvantage of this methodology that a relatively high degree of logical and mathematical proficiency is required.

A second disadvantage is that this method admits the existence of multiple solutions to the same problem. Each different choice of an invariant assertion can lead to a different program structure. The resulting programs do not necessarily portray accurate and consistent models of the problem's environment or its solution. That is, a "correct" program may still have the "wrong" structure.

In spite of these problems, we view Dijkstra's programming calculus design discipline as an encouraging step forward on the road to developing correct programs. It is a method which you should be aware of, for it definitely holds promise for the future.

5.3. Managing the Development Process.

A number of software engineering techniques focus on improving the software development process rather than improving the software developed by the process. They consist of a collage of management and organizatonal strategies plus procedures for integrating and testing the final software product. The techniques discussed here tend to complement the design strategies discussed earlier. That is, given a good design strategy which results in a good program structure, software development techniques exist which can be fruitfully applied to obtain a correct program faster. When these techniques are not used in the context of a good program structure, however, they can become both cumbersome and unmanagable.

5.3.1. Teams.

Programming teams can complement the hierarchical program design with a hierarchical people organization. Given any organization which must work toward a common goal, a certain number of interactions must occur among the people doing the work. As shown in Figure 13 the number of interactions required within a given team is a strong function of team size. As the number of people which must interact gets larger, the number of potential interactions grows quadratically and can take a larger and larger portion of each person's time. Teams of three to five people have generally been regarded as "about right", while teams of 10 to 12 have usually been regarded as "too large".

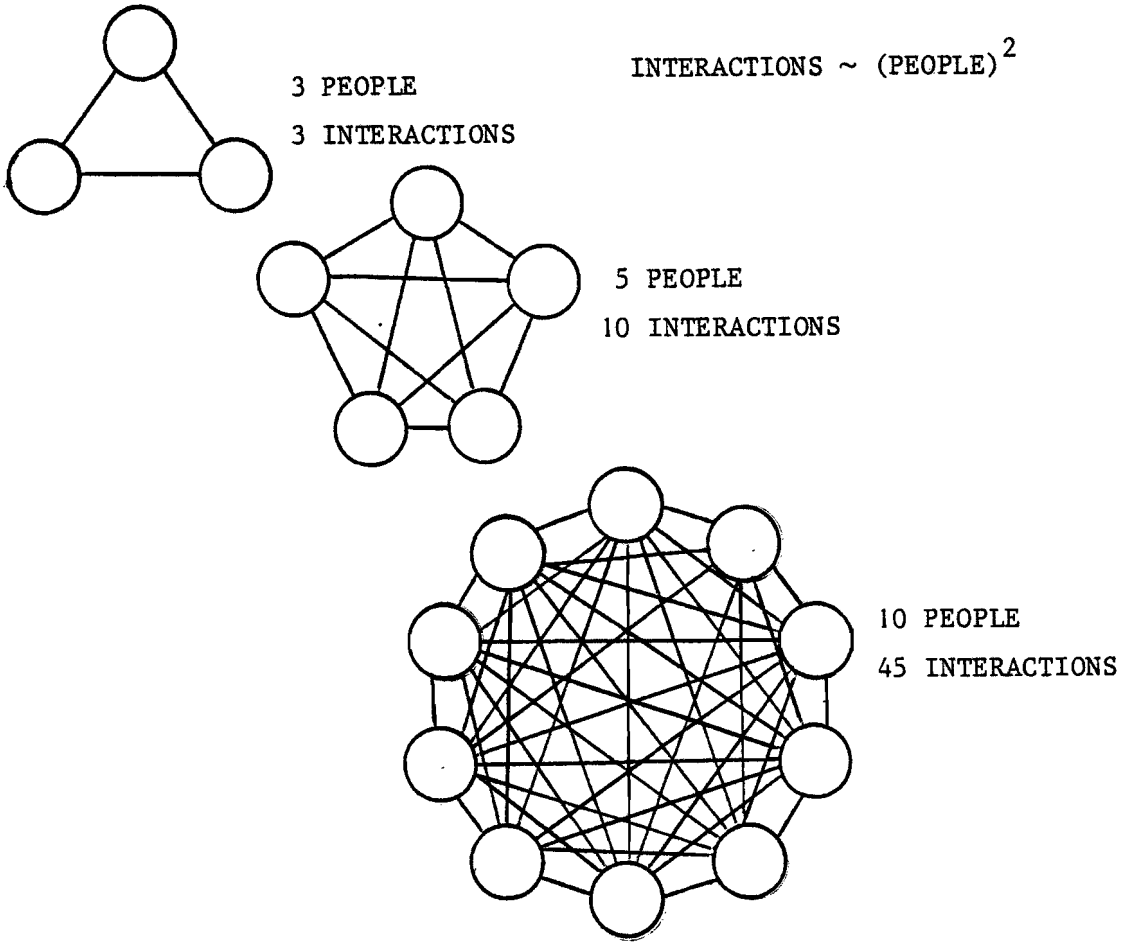

Figure 13. Programming Team Interactions

The concept of assigning programming teams to particular functional modules within a large system localizes most of the people interactions to the team itself and minimizes the communication required between teams. If the program structure has low coupling between modules, the program teams require relatively few interactions with other teams. Conversely, a program module which is coupled tightly to many other modules will place a significant communication burdon on each implementation team.

In most cases, programming teams are staffed with graduate engineers and computer scientists who have long-term career potential with the company. These people are often supported by program librarians or programming secretaries who perform the clerical and administrative functions, freeing up the programmers for solving the technical problems. This partitioning of work is carried even further in a "chief programmer team". Part of the motivation behind chief programmer teams is that the requirements and design part of the job, using structured programming techniques, far overshadowed the coding part of the job. Thus, many people have moved away from the concept of having one designer feed work to three or four less qualified coders. The coding is less than 15 percent of the total effort and in many cases it is easier to do than to delegate. This is especially true when the capabilities of the designer and the implementers differ by nearly an order of magnitude.

While the "chief programmer team" concept sounds nice in theory, most structured programmer teams tend to be teams of peers with the possible exception of a team leader and a program librarian. This team is encouraged to identify as a group with the total job, not individually with pieces of tricky code or a particular part of the job.

In projects where this approach is not taken, an individual is often assigned the responsibility for maintaining a particular segment of code. These people become experts at living with and fielding questions about the idiosyncrasies of their own particular segment of code. In this organizational structure only the person responsible for the code must know how to read the program, so there is no strong incentive to make it understandable to others. A poorly written piece of code tends to stay with its creator for a long period of time. Since changing assignments would require costly apprenticeships, people become relatively immobile. In this environment, comments concerning the quality of a particular program are usually viewed as reflecting on the quality of the programmer.

This contrasts with the concept of having a team responsible for a major program subsystem which tends to allow one to move toward Weinberg's "egoless" programming concept where everybody is expected to know and be able to read all of the programs in their subsystem. The team works toward making all of the programs easy to understand and easy to modify for the sake of both new users and new team members. People outside the team are encouraged and supported in the use of the programs. Since in this environment a single team can carry a new feature from specification through to first application, it tends to provide a good environment a la Ford's "motivation through the work itself".

A concept which has some similarities with egoless teams is called the buddy system. In this system, a group of people is responsible for a group of programs as before, but no one person follows any given program all of the way through the development cycle. A person on the team might design one program, code another, and test a third. Thus, of necessity, a program must be designed and written so that it can be understood by at least two other buddies.

5.3.2. Walkthroughs.

Walkthroughs were one of the first things implemented by most organizations in trying to adopt structured programming. In several large organizations, it quickly seemed as if every hour of every day, some kind of walkthrough was being held by someone. A typical scenario involved some poor programmer trying dutifully to explain his or her program to a dozen or more people who couldn't care less. Soon the size of the audiences diminished, but still is was a painful time for all.

Programming walkthroughs build on having clearly understandable specifications, program designs, and code to walk through. They complement the concept of working in programming teams.

The program walkthrough process is defined in Figure 14 as preparing for and holding a series of meetings in which problems are identified but not resolved. These meetings are normally held when system requirements have been determined, when the system design is completed, and when the code has been written. Walkthroughs are normally called by members of each programming team, with only those people directly involved in the topic at hand being invited. *The major motivation for having walkthroughs is to try to find errors earlier in the development process.* The earlier an error is found, the easier it is to correct, and the less it impacts the rest of the program development process. *A secondary benefit of program walkthroughs results from the interactions within the walkthrough meetings themselves, in which a new kind of informal classroom situation often developes.* In many cases, the older, more experienced programmers can give the benefit of their experience to the newer programmers, and the newer programmers can share the recent advances in computer science which they just learned in school with the older programmers.

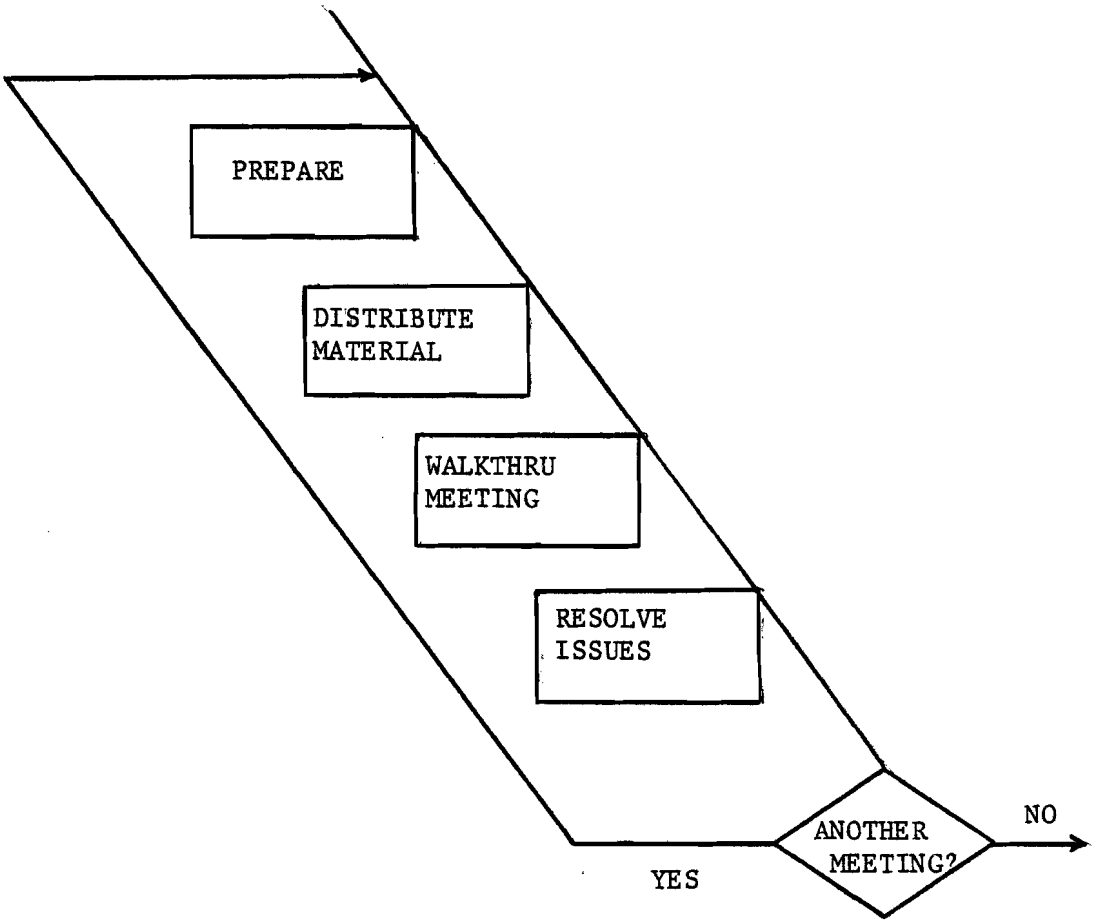

Figure 14. The Walkthrough Process

As noted earlier, there is a potential hazard in holding program walkthroughs if the size of the group attending ends up being quite large. Just going to the meetings associated with the walkthroughs can be a very time consuming activity for the program development team members. One guard against this is to strictly adhere to the rule that problems are only identified and not resolved in the meeting. The second safeguard relies on having a well-structured program. If the program is structured in a fashion which minimizes connectivity and maximizes independence, then the number of people involved in any given program walkthrough will be naturally small.

Given a hierarchical design, the people who have major interfaces with any given program are relatively easy to identify: thus, the number of people involved can be kept relatively small. A program which can be characterized as "spaghetti" coding would require that larger audiences be present at each walkthrough, since any given program is capable of directly affecting a large number of other programs.

Program walkthroughs, management reviews, and project checkpoints all share the aim of trying to find errors early in the development cycle to reduce development costs.

5.3.3. Top-down Implementation.

Top-down implementation and testing and top-down design are completely separate issues. The first procedure concerns order of implementation and the second is a design strategy.

Top-down implementation can reportedly reduce the effort required for program integration and reduce the effort required for program testing. It is a testing and integration philosophy which can capitalize on the modular hierarchical design discussed earlier. Given a hierarchical program design, one can implement the program by starting at the bottom of the hierarchy (that is, by implementing the leaves of the tree structure), writing drivers to exercise these modules, and gradually work one's way up by considering larger and larger subtrees. The problem with this approach is that one often starts by testing and implementing the most detailed, change-prone modules first. Naturally, a series of small changes in the requirements will occur steadily throughout the life of the project. Since the most detailed modules are implemented first, they must be modified the most over the longest period of time. This can lead to severe compatibility problems later, in the program integration phase. It also requires that a significant number of program drivers which are not deliverable software will be produced and debugged and tested along with the main program.

In top-down development the idea is to write and execute the most critical high-level code first and to stub functions which are not yet completed. This is shown schematically in Figure 15. The advantages of this approach are that:

1) No separate drivers are required.
2) The critical programs are tested the most.
3) The code which is the least likely to change is developed first instead of last.

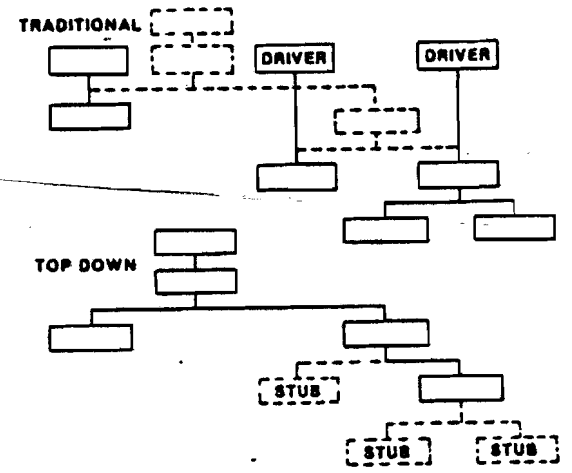

Figure 15. Top-down Implementation.

A related implementation strategy goes under the name "phased builds". With some planning, program modules can be implemented in an order which allows partial processing of certain inputs almost immediately. More complicated inputs are handled in succeeding versions.

5.4. Software Development Tools.

Software development spans many different types of activities, from system requirements analysis through the production, maintenance, and administration of the system itself. When these activities can be organized into a design methodology and a development process that works, the next step is to automate those things which are routine so that more time can be spent on the most creative parts of the design. Design is essentially a problem-solving procedure, and problem solving is by nature a trial-and-error process. By automating the routine parts of the design process, more trials can be made with fewer errors.

5.4.1. Development Support Library.

The concept of a development support library (Figure 16) is a major step toward moving programming from a private art to a public science. It has the advantage of relieving the programmers from the clerical duties associated with programming, and keeps the progress, quality, and interfaces visible to team members and and supervisors throughout the development cycle.

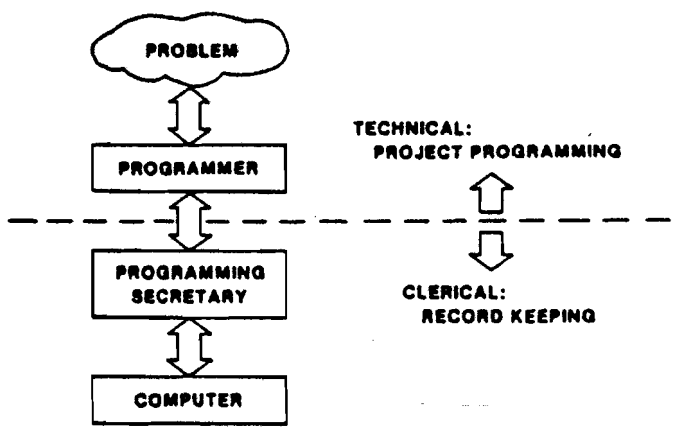

Figure 16. Development Support Library Interface

A successful development support library is based largely on following a set of logicqal, orderly procedures during the development process. These procedures are often aided and enforced by automated support tools for controlling and manipulating the text and code associated with the software system. The intent is to keep the programmer working on the problem to be solved and to automate or at least standardize the procedure for integrating each program into its environment. In many cases, a programming secretary is involved who interfaces the programmer with the development support library. In more sophisticated computer-aided design systems such as the PSL/PSA system many of the clerical, bookkeeping, and documentation functions are performed automatically by the computer.

5.4.2. High-level Languages.

High-level languages have long been recognized as an aid to improving both the quantity and quality of a programmer's output.
A noble long-term objective for high-level language continues to be that of preventing the writing of bad programs.
A practical near-term objective might be that programming languages should at least allow you to write good programs if you choose to. Any higher-level language uses in the context of new software engineering techniques should provide as a minimum those control and data constructs which support structured programming. Proponents of higher-level languages state that the increase in understandability which results allows the programmer to optimize in the large, rather than in the small. Thus, there may be the hope of gaining back any reduction in code efficiency which results from the use of a higher-level language.
Other advantages claimed include the conjecture that even though the best assembly-language programmers can often write significantly more efficient code than the best high-level-language programmers, the programs written by average assembly-language programmers and average high-level-language programmers do not vary greatly.

We have progressed to the point that the most pressing language problems seem to be in the area of *requirement specification languages*. The object is to specify the problem in an easy-to-use, problem-oriented language which is written at a very high level but is still machine executable. This is both a noble and a difficult task.

5.4.3. Documentation.

There are almost as many documentation techniques as there are design methodologies. For discussion, they can be sorted into general categories depending on whether they are primarily text oriented or graphical, on-line or off-line, mechanized or manual, etc.

Text-oriented documentation strategies have the advantage that they can be easily stored and updated on a computer using only a simple text editor. They are also relatively easy to adapt to automatic test generation schemes or program consistency checking. Structured English and pseudo code have the advantage of being able to be stored conveniently and compactly in the same file as the code itself. In cases where the pseudo code is itself a very high-level language which is executable, one can substantially reduce if not eliminate the age old problem of having the documentation and the code out of phase. Unfortunately, however, the closer the documentation format comes to a programming language, the worse it becomes at providing a clear view of the overall program structure. The completeness required by an executable language seems to be at odds with the clarity which a good documentation technique should provide.

At this point in time, there is nearly universal agreement that flowcharts represent an inferior method of documenting a program. This consensus very nearly coincided with the time that automatic flowchard-drawing programs became widely available. An improved form of telescoping sets of hierarchical flow graphs is present in the documentation technique that is called the Structured Analysis and Design Technique (or SADT). While it appears to involve a lot of detailed art-work, much of the effort can be automated. It has found application in both business data processing and real-time control applications.

Graphical techniques usually provide a clearer view of the program but usually suffer from requiring manual updating. This leads to "documenation lag" which can cause no end of trouble.

5.4.4. Structured Testing Aids.

Structured design and development imply a testing effort which proceeds step-by-step in parallel with the application program development effort. This function may or may not be performed by a separate organization. A library of tests which can be applied repeatedly for regression testing, and a library of stubs which stand in for program modules which have yet to be developed, are crucial at each stage of the top-down development process.

When testing is considered as an integral part of the development process, the possibility - indeed, the probability - of finding errors early is increased. In this case the testing philisophy and programs are carried through the specification, design, and coding walkthroughs concurrently with the application programs.

Automated testing has progressed to the point that a "cover set" of tests can be generated automatically which will guarantee that every leg of code is executed at least once. This is of course much different than testing every combination of paths which could be taken through the program, which would be necessary for "exhaustive" testing. Exhaustive testing is just not feasible for many problems of interest.

At this point it should be clear that tests can be written and automated which will give you some degree of confidence in the program if the initial design is done properly. It should also be clear that, in most cases, there is no way that the correctness of a the program can be verified by testing. Exhaustive testing is usually impractical, and without exhaustive testing the correctness of the program cannot be guaranteed.

6. Software Validation.

6.1. Introduction.

The phrase "reliable software" has two distinct shades of meaning. On the one hand, it implies correctness. *A piece of software is correct if it meets its functional specifications*. This implies that as long as the inputs satisfy the specifications, the software will produce the desired outputs. On the other hand, reliable software implies robustness. *A piece of software is robust, or fault tolerant, if it can be expected to deliver a certain minimum level of services even when faces with an unexpected or hostile environment* (such as hardware failure or bad data). Thus a piece of software is reliable if it is both consistent with respect to the stated specifications and able to withstand unexpected demands.

Correctness is a property like beauty that is often mostly in the eye of the beholder. Software almost never turns out to be totally waht the customer had in mind--that is why formal specifications and contracts are needed. A customer may specify that a system with certain functional properties should be built. Because the environment in which it will be used is complex and so well known to the customer, he fails to spell out many of the things that he takes for granted. The resulting system may then turn out incorrect to him. It is because of this possibility of subjective opinions on the correctness of a system that more precise notions of correctness are being developed. We will discuss these approaches below as one of the contributing factors to software reliability.

Robustness is concerned with making programs well-behaved in the face of hardware failure, bad inputs, unexpected demands--even incorrect operation of parts of itself. All software operates in an environment over which the builder of the software has little or no control--hardware fails, other software interfaced to the system works incorrectly, users provide bad input and overload the system. If we are concerned with reliability, we must make our software robust so that it can cope with such situations.

Coping may mean finding alternative ways of carrying out required functions even though something is wrong. It may mean notifying a higher authority that something is wrong. It almost always means not propagating the error so that problems are contained and catastrophe does not ensue. It may mean finding some way to recover from the malfunction.

It is important to stress that correctness alone is not sufficient. A perfectly correct program that does not check inputs to make sure they are withing range may proceed to overwrite valuable files, producing highly unreliable behavior. Certainly a program must be largely correct before we call it reliable, but this is only a necessary condition, not a sufficient one. Robustness is an essential ingredient of reliability.

Software validation is concerned with analyzing software to determine the extent to which it performs the logical functions intended by its creator. Techniques in software validation can be classified into two main categories: testing and verification.

Software *testing* is concerned with analyzing a program by evaluating its response to a selected set of input data. Since any test data will necessarily be a very small sample of the possible inputs, testing is inadequate for achieving a complete understanding of the logical or the performance properties of real programs. However, testing may be a necessary step in program validation not only to clean up certain obvious bugs, but also to reveal unexpected behavior and thus aid a programmer's understanding of his program. It should be pointed out, however, that techniques for testing remain rather ad hoc.

There are two ways of approaching program *verification*. The *static approach* considers a program and its specifications to be given.

Mathematical proofs are developed to demonstrate that the logical behavior of a program is as specified, viewing this logical behavior as completely characterized by a set of formal assertions. The *constructive approach* lays stress on the correct development of a program. However, it seems that both approaches rely on the programmer's ability to abstract certain sufficiently strong invariant properties of the program, intended or given. These properties form the inductive hypothesis that incorporates certain inductions for proving the correctness of loops.

The basic idea of the static approach stems from Floyd's observation that a computer program can be thought of as a mathematical object, and hence its properties can be studied in a rigorous fashion. Floyd shows that it is possible to capture the "invariant" properties of a program at each node of a flowchart program by means of a formal assertion. To show that a program is consistent with its specifications, it is necessary to show that the input assertion implies the output assertion over all possible paths between the start and halt nodes.

As we have seen, the notion of robustness, or fault tolerance, is concerned with providing design redundancy in software such that it can continue to perform, perhaps in a degraded mode, under changing requirements or hostile environments. Providing redundancy is, of course, a common practice in any engineering design. Buildings, bridges, highways, and other constructions, are built with certain tolerance factors. In the last decade there has been a great amount of research into the design of fault tolerant computer hardware. However, research efforts in software fault tolerance began relatively recently.

6.2. Reliability Theory.

There is a well-developed theory of reliability in other engineering areas which has been used extensively in the design of computer hardware. Several researchers are attempting to apply some of these same reliability measures and estimation techniques to software. Our understanding of reliability would be greatly refined if we had precise measures of a system's reliability. The value of being able to estimate a system's future reliability on the basis of past performance is clear.

Like most of the work on software reliability, these formal studies are recent enough that we cannot accurately assess their eventual strength. The fact that software components do not have a failure rate that can be related to the underlying technology in the same way that the failure rate of hardware components can be means that new approaches must be developed. Taken in the aggregate, however, one can study overall failure rates and these may be related to software structure in some cases.

These formal approaches to studying reliability will certainly contribute to our ability of creating reliable systems and must be studied by one seriously concerned with software reliability. However, just as formal approaches to insuring the correctness of a program will never be able to provide us with totally reliable software, formal definitions and estimations of reliability will never be able to characterize completely what we intuitively understand.

6.3. Improving reliability.

In this section we will indicate the range of current efforts aimed at improving software reliability.

6.3.1. Correctness.

Constructive programming is a term applied to any programming method that attempts to produce correct programs without the usual testing and debugging phases. Structured programming, top-down programming, and step-wise refinement are all constructive approaches that in many instances result in programs that are substantially more correct than ones produced in less organized ways.

Testing is the filter that is intended to determine if a program is correct, but often doesn't. Thus, any technique improving the effectiveness of testing will result in more correct programs. Automatic test case generators, performance monitors and automated testing systems all help us improve testing. Specialized testing procedures for particular classes of software can also be developed.

As with constructive programming, many testing techniques and tools are in the development stage. Thus, it is still essential in most shops to make sure that programmers use standard manual techniques for testing.

Static verification techniques are based on mathematical proof procedures and offer the advantage of permitting us to prove the correctness of a program without resorting to testing. Strictly speaking, program verification is a process of proving mathematically whether or not a program will fulfill a set of assertions about its operation when it terminates (which is treated as a separate problem). The assertions are assumed to express our requirements for correct operation of the program so that by proving the program obeys the assertions, we will have proved its correctness.

Unfortunately, there are two problems with this approach which have not been entirely overcome: First, we may make a mistake in stating the assertions so that even though the program is shown to meet them, we may have failed to capture our intuitive understanding of correct operation for the program. Second, the proofs themselves may be quite complicated and thus are open to error if done by hand or are beyond the reach of current automatic methods.

The underlying reliance on mathematical reasoning, however, is closely tied to the original notions of constructive programming as put forth by Dijkstra. One can develop the assertions before actually programming and then write the programs to make the assertions true. A large amount of current research is aimed toward developing proof techniques (both formal and informal) and toward developing practical programming languages that facilitate such proofs. This research is too extensive to review here, but can be expected to provide significantly improved methods of obtaining the programs that are correct (in the broader sense of meeting our expectations, not only our formally stated requirements).

A middle ground between completely formal program proof techniques and completely manual approaches uses formal techniques mated to practical and human-aided tools. This approach may well provide us with the means to produce highly correct programs. Most automated program verification techniques are still a long way from being really practical and most manual or semi-automated techniques are not widely used because they still require a good deal of effort and care to use properly.

This brief review of techniques for constructing correct programs may give you the impression that no one has thought about the problem until recently and that it is all still black magic. While programmers have always tried to build correct programs, it is true that explicit concern with program construction techniques and tools for aiding the process are recent and that for the most part we are only beginning to utilize improved methods.

6.3.2. Robustness.

Improving our ability to build correct programs is a very general task. Improving our ability to build robust programs provides a much more focused goal: We must devise mechanisms that will permit software to cope with whatever unexpected occurrences threaten its operation. Some of the reliability mechanisms are really no more than just good programming practices that have been used for years: checksums, checking of parameter ranges, data validity checks, and so on. Yet, the sad fact is that many programmers do not use such mechanisms and there are really no good "handbooks" one can consult for techniques. Thus, until an encylopedic treatment of reliability appears, you must laboriously gather good mechanisms from the descriptions provided by others or else invent your own.

Let us mention just a few here to illustrate the type of structures that provide robustness. *Self-identifying information structures* provide protection against some types of hardware failure or the accidental destruction of pointers to the structure. The simplest example is a disk file that includes a header containing the name and other identifying information of the file; if the directory that points to the file is destroyed, it can be recreated from the information in the file itself.

Modularity provides an opportunity for improving robustness. As data and parameters are passed between modules, the opportunity exists to check them agains expected values and thus detect problems before they propagate. This and the following mechanisms provide what are often called firewalls.

Many of the mechanisms developed for making systems secure, also provide increased reliability. When resources are carefully controlled, for example, checks of resource ownership will be performed. This provides an opportunity to catch erroneous (not just illegal) system operation through the detection of internal errors.

A new structure developed specifically to provide robustness is the idea of *recovery blocks*. Simply, this is a menas of indicating portions of a system whose operation must pass a dynamic acceptance test designed to determine proper operation; if the test fails, alternate means of achieving the desired result are indicated and automatically tried. (This mechanism obviously cannot be applied everywhere since it assumes a computation can be retried.) As structures providing robustness are developed, the task of making systems reliable will become correspondingly easier. The critical role of design in providing overall reliability will remain.

6.4. Trends in Software Design.

Software design techniques are undergoing active investigation and change. While these changes are only beginning to affect significant numbers of people, several important trends that will directly affect reliability can be identified.

In our following consideration of trends, bear in mind that we here are most concerned with the impact of design on reliability.

6.4.1. More Design, Less Coding.

The most profound and widespread trend is the increasing concern with the specification and design process. People are realizing that the difficulties experienced in creating large software systems are not due solely to bad programming practice or insufficient management.

It is true that it is often possible to create a small program without engaging in much formal design activity. A craftsman (programmer) can usually create an acceptable small program without engaging in much documented consideration of alternatives (i.e., design).

We might compare this to craft production of artifacts in other areas. For example, a boat builder may create a small boat similar to ones produced before, but with some variation, without drawing detailed blueprints. But a shipbuilder cannot build an aircraft carrier successfully without engaging in a large amount of design--consideration of alternatives, careful fitting of form to constraints, checking that all that is required has been included, and planning of the actual implementation.

We now realize that the same is true of building large software systems (and sometimes even small ones of high complexity). The tasks we have set for ourselves demand design skills and techniques beyond the valuable new programming methodologies, such as structured programming.

One result has been increased awareness of the important role that design must have. As people have become concerned with properties of software, such as reliability, then it has become even clearer that these properties must be designed into a system from the start and that trying to add them on at the programming stage will not work.

The primary impact on software reliability that this increased emphasis on design will have is to provide an opportunity in the software creation cycle for proper consideration of reliability. Although reliability is our interst here, it should be clear that creating a software system involves many factors: (e.g., generality, portability, and maintainability) which may deserve careful consideration; acceptance of the software by the eventual user must be considered and often goes beyond technical qualities in importance; and economic factors typically cover all other considerations in the sense that for any given property (such as reliability), the amount of it we get is usually a direct function of how much we are willing to pay.

When software is slapped together with most of the time spent on testing and retrofiting brought about by precipitious coding, there is little opportunity for considering the inevitable trade-offs between various qualities and for making sure that features that must be dealt with on a system-wide basis (e.g., reliability) are considered. The hallmark of design is the consideration of alternatives and the weighing of conflicting demands to find acceptable compromises. As we move toward more extensive design, we at least will have the opportunity to consider reliability and to do it on a system-wide basis.

6.4.2. Coherent Methodologies.

The current interest in software design focuses on two areas: developing coherent methodologies for design and developing tools to aid or argument the designer. We will briefly address this second focus in a later section. *A methodology is a collection of methods (techniques, procedures) to follow, which if faithfully carried out and applied in a particular situation will (in theory) achieve some goal (e.g., the attainment of a correct design).* A methodology is more than a recipe. It usually consists of several aspects, may not be complete, and it typically cannot be applied blindly.

For example, a design methodology might prescribe the order in which certain classes of decisions are to be made, ways of making decisions, ways to represent the developing design, and so on, but not address project management. The assertion is then that if the prescriptions are followed one will be able to achieve a correct design more easily.

We have seen a number of methods, but they hardly form a methodology.

The impact of a coherent methodology should be clear: Producing a reliable system is an activity that must be spread over the complete creation process. When that process is composed of mismatched techniques and when the overall management controls permit some phases to be missed then producing reliable software will be next to impossible. Coherent methodologies eventually offer solutions to these organizational causes of unreliable software.

6.4.3. Modularization.

Breaking a program or system into pieces, or modules, is one of the oldest concepts in computing. Yet, it is also one of the current trends in software design, as we have seen.
The definition of what constitutes a module is now seen as an important design parameter. It is clear that modularization is a necessary condition for pragmatic program verification.
Whereas previously modularization was often simply used to break a taks into pieces small enough for a single person to work on or small enough to fit into a memory space, we now see that modularization will have a very real effect on our ability to understand and deal with the complexity in a system.
Modularization has a very direct impact on program correctness by reducing complexity into manageable units and thus reducing the chances of error. Further, modularity permits application of formal verification techniques to programs small enough that the proofs are manageable.
The impact on robustness can be gained through the use of components. Small, highly robust modules can be developed and then reused in other systems. Since they are known to be reliable, the task of producing a new reliable system is reduced.

6.4.4. Formal Specifications.

A fourth trend in software design, not yet so well developed but receiving a large amount of attention, is the use of formal specifications. Earlier we noted the difficulty of assessing the correctness of software in many instances because it is not clear what the software is supposed to do. In response to this some have attempted to develop mor explicit ways of specifying the goals for a piece of software.

The Problem Statement Language (PSL) used in ISDOS is a language for stating software requirements in an unambiguous and functional manner. More formal approaches, typically using mathematical logic to state requirements, have been tried but have not yet been turned into practical tools. The assertion languages (again, often based on predicate calculus) used to state correctness assertions for use in program verification are a way of formally specifying what a program is to do.

Parnas has developed a technique for specifying modules and this technique has been coupled with a comprehensive design methodology by a group at SRI to attack a significant sized problem (the design of a secure operating system).

The primary impact of formal specification will be on correctness. Not only will such techniques, when further developed, make it easier to check formally the correctness of a system, but the existence of clear and unambiguous specifications will help focus the efforts of the designers, resulting in designs of better quality.

6.4.5. Design Verification.

A trend only beginning is that of design verification. The idea is simple and in fact has been practiced for many years: We would like to test that a design is correct before we turn it into the actual object. Design reviews are intended to do this. But, they don't fully succeed because the complexity of software design decisions makes it impossible to verify designs with any accuracy using informal means.

Techniques that permit us to verify the correctness of a design with some accuracy will have a great impact on reliability. Not only will they help us to improve vastly the correctness of our software, but they may help us see the results of various possibilities before building the system. This wil permit us to build in mechanisms to provide robustness on the basis of this early feedback concerning possible vulnerabilities.

6.4.6. Metacode representation.

We noted above the importance of representation in software design since in designing, we are solely building a representation, not the final object. A very valuable trend in software design is to use a programming language-like notation in which to express a software design, especially at the levels of most detail. These notations are usually called a metacode or program description language (PDL).

Most metacodes use the textual structure and control mechanisms of the base language while permitting more freedam in the expression of conditions and operations. This freedom may range from completely unrestrained use of English to highly constrained use of predefined and mnemonically named functons.

Metacode makes visible many of the logical decisions concerning control and data before the lowest level programming details are added. This permits the designer to check the design for logical correctness more easily since the metacode is usually much easier to comprehend than a full programming-language representation. This makes it easier not only to achieve correctness, but also to assess the robustness of the system.

Because the metacode has the same structure as a programming language, most people find it extremely easy to code from it. Basically, the use of metacode has the effect of pushing detailed design and debugging up a level into a representation that facilitates, rather than hinders, obtaining correct and robust programs.

6.5. Automation of Software Deveopment.

Several areas of software design with special relevance to achieving reliability that could benefit greatly from automated aids are:

Processing of representation: The design of large systems involves a large amount of information. Designers should be relieved of burdensome clerical duties and their activities enhanced by simple aids such as cross-indexing and syntax checking of designs. This will permit them to focus more on the content of the design.

Checking of consistency and completeness: Usage of the same information in different places in a design should be consistent. Routines called as subroutines with particular parameters should be checked to make sure they are set up to handle those parameters. Functions or data structures used in one part of a design must be specified elsewhere. These and other questions of completeness and consistency can be checked automatically in many instances.

Comparison to constraints: Design is a process of creating an artifact within constraints. If those constraints are formally stated then a developing design, properly represented, could be checked automatically for conformity. In large systems with many detailed constraints, such automated checking is essential.

Comparison to specifications: Again, as we develop formal means of specifying the functions of a system, techniques can be developed for automatically checking a proposed design against the specifications it is supposed to meet.

Suggestion of alternatives and ramifications: Another fundamental characteristic of design is the choice between alternatives. If a design is developed in a machineable representation and if one has built up a library of possible structures, then a design-aid system could suggest alternatives to the designer. This would be especially applicable to providing suggestions with known reliability properties. Additionally, when design decisions are made, an automated monitor could determine if anything is known about the characteristics of this structure (e.g. its susceptibility to error) and suggest ramifications of the decision to the designer.

These brief descriptions of areas in which automation and/or augmentation look reasonable should suffice to indicate ways in which trends in software design may impact our ability to build reliable systems.

7. The Choice of New Software Development Methodologies.

7.1. Introduction.

The data processing project manager of the 1980's has an impressive array of new "structured" methodologies which promise to improve the productivity of his programmers and analysts, as well as improving the reliability, maintainability and overall quality of the finished product.

Unfortunately, there are so many "new" methodologies that the manager may not know which methodology he should employ on a new project. That choice is made all the more difficult because there is little or no documented evidence to prove the effectiveness of the new methodologies. Indeed, the problem is even worse: a variety of exaggerated claims in the popular EDP trade journals has made many a manager so skeptical that he may be unwilling to experiment with any of the new methodologies.

The purpose of this chapter is to provide some useful advice to the project manager who finds himself in this position.

7.2. Suggestions for Introducing the New Methodologies.

Unfortunately, it is not possible to give a simple algorithm in this area. We cannot easily say: "First you should introduce structured programming, then you should use structured design", nor can we say: "If you are working on a payroll system, then you should definitely use chief programmer teams; on the other hand if you are developing a real-time telecommunications system, you should use only structured walkthroughs."

On the other hand, the structured methodologies have been introduced into enough organizations that we can draw some general conclusions form their experiences. These are given below.

Trying to implement all of the new structured methodologies at once will generally be a disaster.

Some organizations can actually pull off such a feat. After reading about the new methodologies, or getting a presentation form their friendly hardware vendor, they decide to use all of the new methodologies at once. As one might expect, this is more likely to happen in the smaller EDP organizations--those with only half a dozen programmer/analysts--and is not verly likely to occur in the larger organizations.

Sometimes, though, an organization will decide to try all of the new structured methodologies on a single project; this is quite common when the organization decides to use the new methodologies as an experiment in a so-called pilot project. Even in a limited situation like this, it ususally turns out that an attempt to experiment with half a dozen new methodologies at once leads to chaos and confusion. The reasons are obvious enough. The new methodologies use no simple concepts, and a lot of concentration is needed to make them work right. If the programmers are also trying to implement walk-throughs--which require a great deal of phychological energy, too--and chief programmer teams, as well as adjusting to the concept of a librarian relieving them of their clerical work ... well, it will be a wonder if they get any of it right.

Techniques which involve organizational change are often the most difficult to implement.

Some organizations will find it difficult to ever implement chief programmer teams, librarians and walkthroughs. The point here is that even if the project manager can convince his organization to try the chief programmer team concept, or librarians, or walk-throughs, he will probably find that difficult as his first new methodology. The experience has been that it is somewhat easier to introduce a relatively innocuous technical concept like structured programming first--that doesn't threaten anyone's empire, and is not likely to be at odds with current organizational philosophies. Once the project manager had demonstrated that structured programming, top-down implementations and structured design are good ideas, then he'll probably be in a strong enough political position to say to the big boss: "Listen, the last three structured methodologies that I introduced to the company turned out to be winners. Why not gamble a little now, and let me try something like the chief programmer team?"

Structured code without a design methodology is often worthless.

A number of organizations have found recently that structured programming (or, more specifically, structured coding) is a great idea but that is not enough. If the modules in an EDP system are too large, too complex, and too interconnected with one another, then maintenance problems will persist.

This raises some interesting political consequences. If the EDP organization has been doing things in a backwards fashion for years, and if the project manager introduces the new methodologies with great fanfare and promises of spectacular improvements, then the first new methodology should indeed demonstrate spectacular improvements.

And if the project manager tries structured programming alone, he might not achieve such spectacular improvements. The experience on a few EDP projects has been that the initial productivity and reliability will seem quite impressive, but the long-term maintainability of a system produced with nothing more than structured coding may not be very impressive at all.

The moral: It may make good sense to begin with a design method first--and when that is working properly, then introduce structured coding. Once the project manager has overcome all of the objections and battles and problems associated with design, it will be almost trivial to introduce structured programming.

There is a more important reason for this suggestion: good design and mediocre coding is a tolerable state of affairs; mediocre design and good coding, on the other hand, is not a good formula for success. And if the project manager thinks that his project team has energy, intelligence and enthusiasm to tackle only one new methodology, then design should get preference over coding.

Top-down design and implementation are often a good way of introducing the new structured methodologies.

It is frequently observed that many of the benefits of top-down implementation are "political" in nature. It allows the project manager to demonstrate a working subset of his system to the user at an earlier point in time; it allows him to survive deadline crises more gracefully; and it allows him to schedule testing resources (e.g., computer test time) in a more manageable fashion.
These benefits are very noticable to the user community, to higher levels of management, to the computer operations manager, and to various other people in the organization. For that reason alone, many EDP managers have decided that the top-down approach is a good way to introduce the new structured methodologies in their organization.
Keep in mind that this approach can backfire. Unfortunately, many programmers view top-down implementation as an invitation to begin coding before they have done any real design. Especially on the first new projects, the manager should beware of this danger.

The most successful approach has often been informal walkthroughs.

There is a strong argument for informal walkthroughs as the project manager's first venture into the new structured methodologies. Note the emphasis on "informal" walkthroughs--not necessarily with all the "bells and whistles" that are normally suggested.
Why would informal walkthroughs be a good way to get started with the new structured methodologies? For the simple reason that the project manager can't trust any individual programmer to understand and implement any of the other methodologies by himself. By forcing everyone to talk about their designs and their code--in an informal, low-key, non-threatening fashion--the manager can maintain some kind of quality control when he most needs it.

This is a point that needs emphasizing. If the manager has 30 programmers, and if he gives them all the standard textbooks on structured programming, they are almost guaranteed to read 30 different (and almost mutually exclusive) things. They will write 30 different kinds of structured programming--some good, some mediocre, and some downright bad (indeed, probably even worse than the kind of code that was written before structured programming came along). And if nobody looks at their code (which is the current state of affairs), the manager will never know who really understands structured programming, and who doesn't.

If the project manager begins by establishing an environment of exposing everyone's code to public discussion, then he will ensure that a relatively uniform version of the method can be implemented later on.

7.3. Conclusion.

In the final analysis, only the project manager can decide which of the new methodologies he wants to introduce on a project. The suggestions in this chapter can do nothing more that make the manager think about trade-offs that have been observed in other EDP projects; it is up to the manager to apply those trade-offs in his own project.

One of the most important questions the manager must ask himself is whether the new methodologies should be considered as a set of experimental "R&D" concepts, or whether they are to be considered down-to-earth practical concepts, with an immediate payoff.

Indeed, some organizations deliberately use the new methodologies on experimental "pilot" projects, with no preconceived ideas about which ones will work and which ones won't. In such an environment, the manager should use any and all of the methodologies that are of interest to him; our only caution is to arrange the pilot project in such a way that the impact of each new methodology can be measured in some crude fashion.

If the manager is involved in a "real" project--with real deadlines, real budgets, real users with real needs, and real penalties if the project fails--then he should be considerably more cautious about the new methodologies he employs.