6

Software Requirements for Computer Integrated Manufacturing Including Computer Aided Software Engineering

ESSENTIAL ASPECTS OF SOFTWARE DEVELOPMENT [48]

THE INGREDIENTS OF A SUCCESSFUL SOFTWARE PROJECT

The following list comprises some of the important aspects of software development, which have to be considered if a project is to be successful:

1. Technology
   - A. The Design
     - 1) Adequacy
     - 2) Modularity
     - 3) Adaptability
   - B. The Software Development Environment
     - 1) Specification Tools
     - 2) Programming Languages
     - 3) Test and Verification
     - 4) Documentation
   - C. Support Hardware
     - 1) Development Machines
     - 2) Special Test Environment
2. Management
   - A. Organization
     - 1) Project Phases
     - 2) Planning
     - 3) Cost Estimation
     - 4) Teams and Structures
     - 5) Influencing Factors
   - B. Human Factors

C. Support Tools

   1) Documentation

   2) Reporting

   3) Checkpoints

It should always be borne in mind that the technological and the management aspects are of equal importance. It is also a fact that most of these topics have been individually investigated and are well known and covered by literature, courses, etc. But obviously it is not yet common knowledge that they have to all together form a "management system" and that in general they are interrelated.

This can be exemplified by the development of the phase model as given below.

THE PHASE MODEL

The phase model was originally derived from management considerations. It later turned out to be a useful framework for the construction and classification of tools. For some time it was even considered as technological dogma. However, people now understand that both aspects (managerial and technological) are interwoven and interdependent.

It has recently been confirmed that an overly rigid phase planning is counterproductive, but that a reasonably phased structure for a project is necessary and useful. It can be stressed that the usual phase plan has to be extended by a phase of thorough planning. The recommended phase model therefore looks approximately as follows [67]:

1. Planning and establishment of the management structure
2. Establishment of quality assurance mechanisms
3. Definition of the requirements
4. Design specification
5. Design and coding
6. Unit test
7. Integration
8. System test and validation
9. Maintenance

THE IMPORTANCE OF GOOD DESIGN

With good reason the design has been mentioned first on the list in the previous section. In principle it should be a matter of self-understanding that for the development of computer and software systems good design is as important as it is for any other technical product. The best tools and the most capable manager can not save a project whose product design is bad or even wrong. Until now software development has been regarded mainly as an aspect of the development (or production) environment, i.e., programming languages, specification tools, test and verification, and, documentation. Such a view would appear utterly strange to the usual plant engineer. Of course the engineer also has to think about the tools with which to produce the design of, for example, a car but primarily is concerned with designing a good and affordable car. The production facilities are then constructed according to the requirements of the product and the company's financial considerations.

But unfortunately very little is known about what comprises a really good software design! Besides, many believe that this problem can not be solved by software specialists alone. In the first place the quality of software is determined by the properties and the requirements of the application. To stay within the above mentioned example of automobile design, knowledge about production methods may help to make a car cheaper, or, perhaps less prone to rust, but it will certainly not improve its road-holding or its fuel consumption. So the manager will have to apply several criteria against which to check the quality of a design. The most important ones seem to be those listed below.

In the first place the design has to be *adequate to the problem.* It must be neither too futuristic nor overly conservative. One must not take unnecessary development risks by trying an unknown problem solution on, for example, a new generation of computer. But one must also avoid "obsolescence on delivery".

Then a design must be *modular*. This is important for technical reasons as well as for organizational ones. From a technical point of view it is well known that a modular system is easier to design, to understand and to maintain than a monolithic one. Under managerial aspects it is necessary to prepare for the necessity to develop a system using a team, i.e., to be able to assign well separated work modules to different people or different subgroups.

Finally a design has to be *adaptable to change*. This does not only relate to changes "after delivery", which Parnas may have had in mind when he postulated his "Design for Change", but also with changes which will, inevitably, occur even in the development phase. This is inevitable because software projects usually take much longer than everyone expects. Everybody talks about an "innovation rate" which is supposed to be between 2-3 years, but statistics teach us that the average software project of nontrivial size takes between 5-8 years! To appreciate this consider that the average lifetime of a government is usually about four years. Thus one may well face drastic changes of the social or political environment in the middle of the development phase.

ORGANIZATIONAL ASPECTS

PLANNING

Everybody agrees that planning is necessary, and in every project it is done at least to some degree. But some mistakes are quite common.

Firstly, planning is obviously not taken seriously enough. This observation has already been described in the book, *The Mythical Man-Month*, by F. Brooks [36]. Brooks describes how project teams are usually built up too fast and that planning is regarded as a kind of "side-activity" for the manager during the early project phases. Instead the bulk of the manager's time is consumed in instructing all the new people and in assigning work packages to them. Consequently these assignments are only partially thought out and are often incoherent because their planning has not been completed. From this an important rule can be derived: *Do not start a software project of non-trivial size with a fully staffed team, but allow for a planning phase, in which a few - but very good - people prepare the project by thorough planning and architectural design!*

Secondly, planning tools are not used properly. They are either not used at all or to the contrary - adhered to too strictly. This, in turn, leads to inevitable frustrations and to abandoning them after some time. It is generally agreed that it is better to use planning tools than to work without them, but that they should only be loosely connected to the project and used as "guidelines" and "early warning systems". So, for example, PERT-diagrams are not rated very highly, because they require too much detail and are difficult to adapt. Bar-charts or Gantt charts are, on the other hand, generally regarded as very helpful.

COST ESTIMATION

Productivity and Cost Models

This is generally regarded as one of the most important issues in the management of software projects. In the USA it has been discussed in conferences for many years and there is a considerable body of literature dealing with this subject. A number of "cost-models" have been developed which try to take into account as many influencing factors as reasonably possible. Therefore many of these have sometimes become quite complex. Despite the effort expended none of them has succeeded in really giving precise and reliable forecasts.

A "rule-of-thumb" can be developed from a comparison of these cost-models. This is, "estimate the possible size of the code in your project and divide it by the productivity of our team". This yields almost exactly the mean value of the forecasts given by a number of more or less complicated cost-models. The rule-of-thumb performs even better if one applies the usual statistical error boundaries to the estimates of code size. Of course it is common knowledge by now that a linear relationship does not hold between code size and project cost for very large software projects, but the cost models did not do any better there.

This difficulty can be overcome by a modular design with loosely coupled components. The explanation for this can be found in the work of Halstead [59], who had discovered that the effort - which is expressed by cost - for the development of a piece of software is not a linear function of the

size of the software, but grows according to some exponential relation. The reason for this is that the true cause for the necessary effort is the internal complexity of the software, which also grows exponentially with the project size. Halstead also showed that modularization can drastically reduce the necessary effort, because the total effort necessary for some large software systems can now be computed as the sum of the effort necessary for all the modules considered together instead of the exponential result one would obtain from applying his formulas to the whole piece of software as a single entity.

Another important principle, which was first described in great detail by Brooks [36], but which is forgotten every time a project becomes critical, is: *"Adding manpower to a late project makes it later"*, or more generally: There is an optimal team size which must not be exceeded if the project is to be completed in a reasonable time. The reason for this is that humans, who work together in a team, have to communicate in order to get the common work properly done. This communication takes time and this use of time decreases the "productivity" (e.g., measured in lines of code per man-year). But as it is clearly impossible to realize a 100 man-year project by one person who is allowed to work 100 years, one has to allow for these "communication losses". But one also has to know that they exist and thus organize the team in such a way that they do not exceed a tolerable amount of the total time budget. Modern cost models obviously take this into account, as Figure 6-1 shows. This figure is taken from [67] and has been computed using the SLIM model [91].

An important aspect of this figure is described by M. Key [67] as follows: "It also shows an Impossible Region. Faced with this evidence it is more difficult for the senior manager to say: 'Well, if you can't do it, I will find someone who can!' Clearly, management must attempt to achieve a required completion date as determined by a market window; what it must not do is go into the 'impossible' region of the graph in an attempt to do this! *Therefore, the plans must be realistic in their time scales and have a degree of flexibility which can accommodate slip."*

But Figure 6-1 also shows another, very important, aspect. From the manpower curves one can see how to do the same work with much less effort just by allowing for a little more time! For example, as shown in Figure 6-1, one can produce 250 K of software using 25 man-years or 100 man-years. In the latter case one has even slipped slightly into the impossible region, i.e., it will be a very difficult project. The resulting saving of time is shown to be less that 30%, whereas from a naive point of view one would have expected 75%. This observation is confirmed by Figure 6-2, which has been taken from a study by IBM [88].

If in Figure 6-2 one locates the team sizes which result from the above figures, i.e., from either approximately 6, or 33 people, on the curve for FORTRAN (empty circles), and looks at the resulting values for productivity and project duration, the values of Figure 6-1 are confirmed: the productivity of an individual in a team of 6 is four times that of the same one in a team of 33, and the gain in project time is approximately 30%. Thus one obviously has detected a rather solid "law of nature". This law was also first described by Brooks, who also found an explanation for it: It is the time for communication between people which is necessary in a team! He also gave a formula describing this effect in quantitative terms. This formula and some of its results are plotted in Figure 6-3, which illustrates in a dramatic way one of the central problems of the management of programming teams.

Even with the modest amount of 1% of time for one team member to be talking to another one, the optimal team size is as small as 6 - 8! Even with this team size there are between 15 - 28 "communications" per person per week. This consumes from 6 to approximately 11 hours per week of *each* person's time. Brooks concludes that, as you can neither forbid communication completely nor have every project team conducted by just one person, one has to *organize communication.* He describes several methods for this purpose in his book [36]. Of course a general caution should be applied in this case as well as in every other one: *Do not try to adapt other people's methods or experience to your problems without reflection and proper adaptation!*

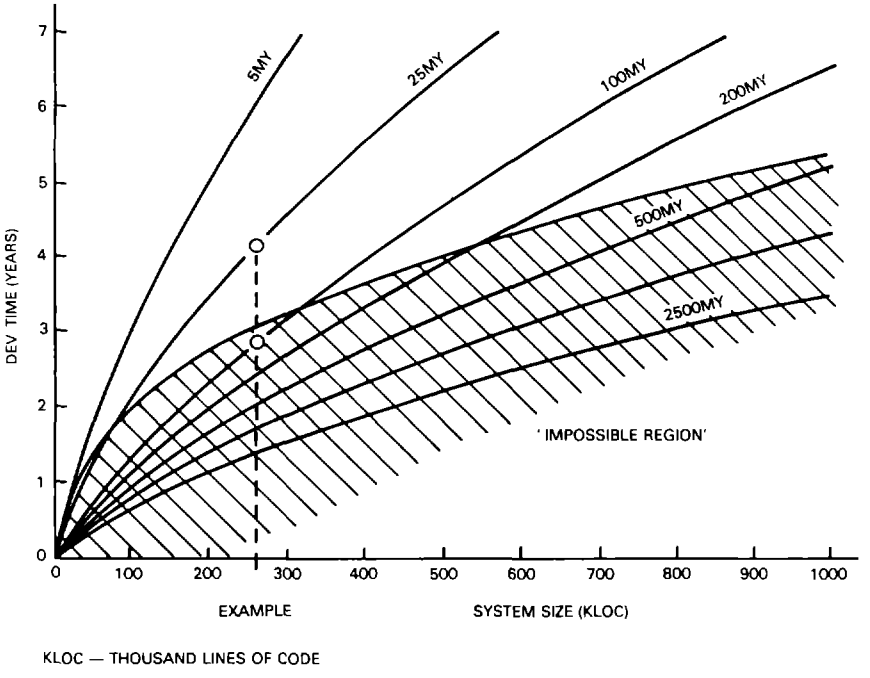

Figure 6-1 Slim—diagram [67].

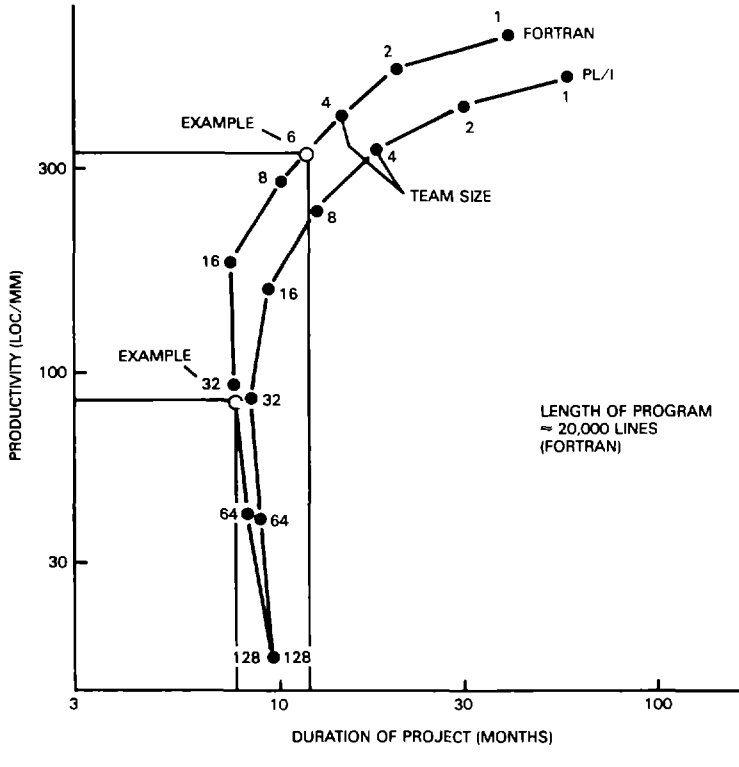

Figure 6-2 Dependence of productivity on team size (adapted from [88]).

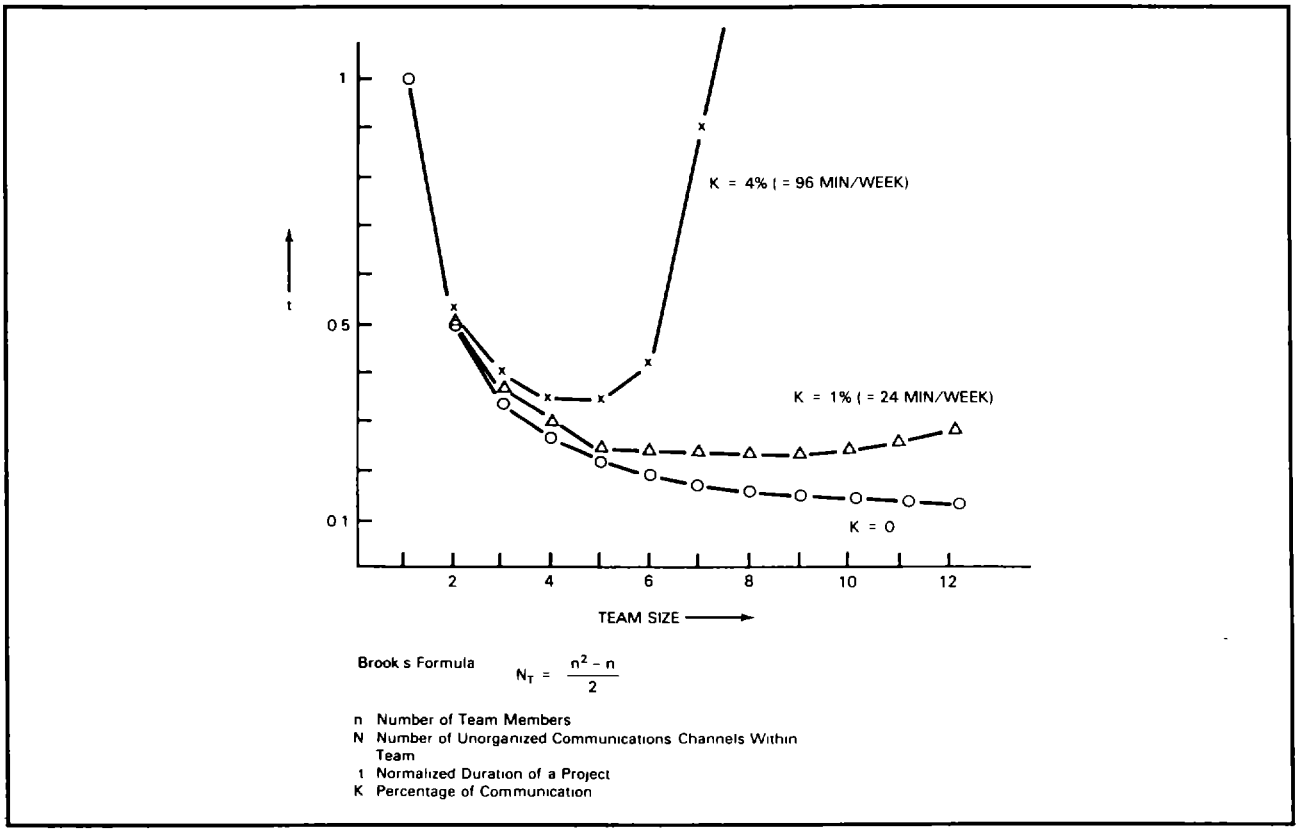

Figure 6-3 Dependence of project duration on team size.

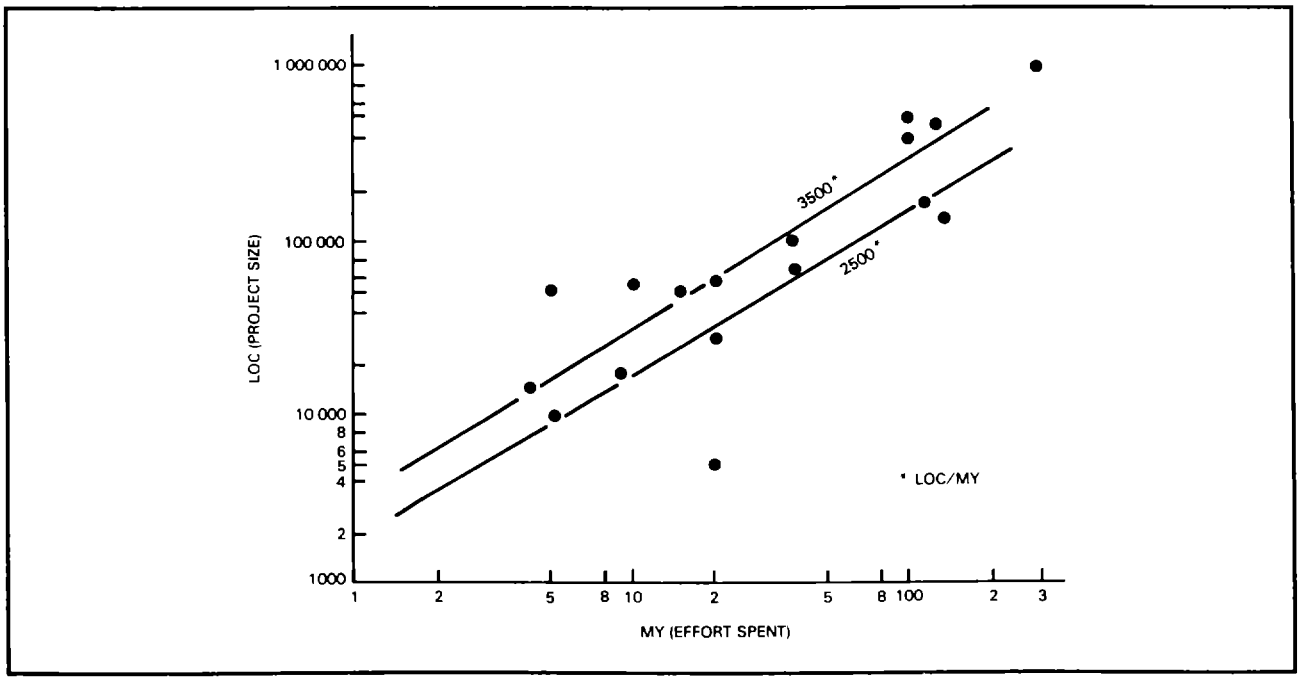

Figure 6-4 Productivity values collected at first IFAC Workshop on SW-Project-Management.

But even if one has thoroughly understood the problems connected with the management of sizable teams and on top of this is a gifted "leader" who really can get people to work, the problem of a reliable original estimate of the costs of a project remains. Obviously really reliable data on programmer productivity do not yet exist. The manager normally has to take recourse to their own experience.

There are possibilities to check this experience for plausibility and to compare the performance of one's own team or company with the outside world. First, one can browse through the published literature for figures, one can talk to colleagues and one can calculate backwards from competitor's prices and/or project durations. But there is also compiled material available: Nearly everything which Barry Boehm publishes (e.g., [32,33]) contains valuable figures and reference information. A less widely known but extremely valuable book has turned out to be a really invaluable source of raw data. This is Montgomery Phister's *Data Processing, Technology and Economics* [88]. This book contains innumerable statistics, collected over a period of approximately 15 years and covers all aspects of data processing from computer production to program development. In addition it is updated at regular intervals.

Figure 6-4 is a plot of productivity figures collected by means of a questionnaire during the Heidelberg workshop on software engineering [48].

The bandwidth of the results corresponds very well with values obtained from other sources:

2500 - 3500 LOC/MY	Workshop average 1986
4000 LOC/MY	Author's own experience, difficult FORTRAN code, 1984
2000 LOC/MY	Author's own experience, difficult Assembler code, 1982
3200 LOC/MY	[111], 1977

(A later, more thorough evaluation of the workshop results showed a wider distribution: 2700 900 LOC/MY) (LOC/MY - Lines of code per man year).

Influencing Factors

The above figures cannot be applied uncritically and universally. One also has to take into account the most important factors which influence the productivity of program designers. The most complete collection and evaluation of such factors can also be found in [111]. There 30 influencing factors have been listed and their effect evaluated. Those with the highest values have been listed below:

1. Complexity of customer interface
   4.0/1.0
2. Experience with programming language
   1.0/3.2
3. General qualification of personnel
   1.0/3.2
4. Experience with application
   1.0/2.8
5. Designer participation in specification
   1.0/2.6
6. User participation in requ. def.
   2.4/1.0 (!)
7. Experience with computer used
   1.0/2.1
8. Complexity of application algorithm
   2.1/1.0
9. Percentage of delivered code
   1.0/2.1/1.7 (!)
10. Limitations of working memory
    2.1/1.0

Some other factors, which usually enjoy a high favor among theorists, are of less influence than expected:

29 Complexity of control flow	1.4/1.0(!)
30 Module size	1.25/1.0/1.35

The figures are an indication of productivity and are to be read as follows: The first figure holds if the respective factor is smaller than normal, the last one, if it is greater than normal. The middle figure (if given) describes productivity under normal considerations. Thus very complex relations with a customer, i.e., something which depends on a talent for negotiations and on the quality of the contracting department, can decrease productivity to a quarter of a good value! On the other hand, if one can assemble a team of qualified people who are familiar with the application and with the programming language (factors 2, 3 and 4), one theoretically has a chance to complete a given project 25 times as fast as under adverse conditions. The purely technological factors, i.e., Factors 29 and 30 are of remarkably small influence. The effect of Factors 6, 9 and 29 is counter intuitive, i.e., experience and statistics show different results from what has always been expected from theoretical discussions.

In general this list easily explains why the reported productivity figures of programmers can vary by a factor of 20! And for a manager, who wants to do reliable planning, this means: *Observe your team, keep your own statistics, monitor your influencing factors and apply a reasonable safety margin in your estimates!*

D. Martin [78] also described and evaluated the influencing factors which had been relevant to his projects. Though he did this only qualitatively, the results have confirmed the values given in the above list.

Distribution of Effort Over Project Duration

As already mentioned above, one should never start a sizable project with a fully staffed team. But what, then, is a reasonable distribution of manpower over the duration of a project?

One curve actually observed in a successful project is shown in Figure 6-5 which is taken from [78]. It shows that a successful Project of over 100 man-years has actually been prepared by two people over one year! A more qualitative approach is used in Figure 6-6. This diagram, however, illustrates the reasons for such a curve by indicating the order and the overlapping of activities in a software project. Less detailed, but supported by good statistics, are the values given in [88]:

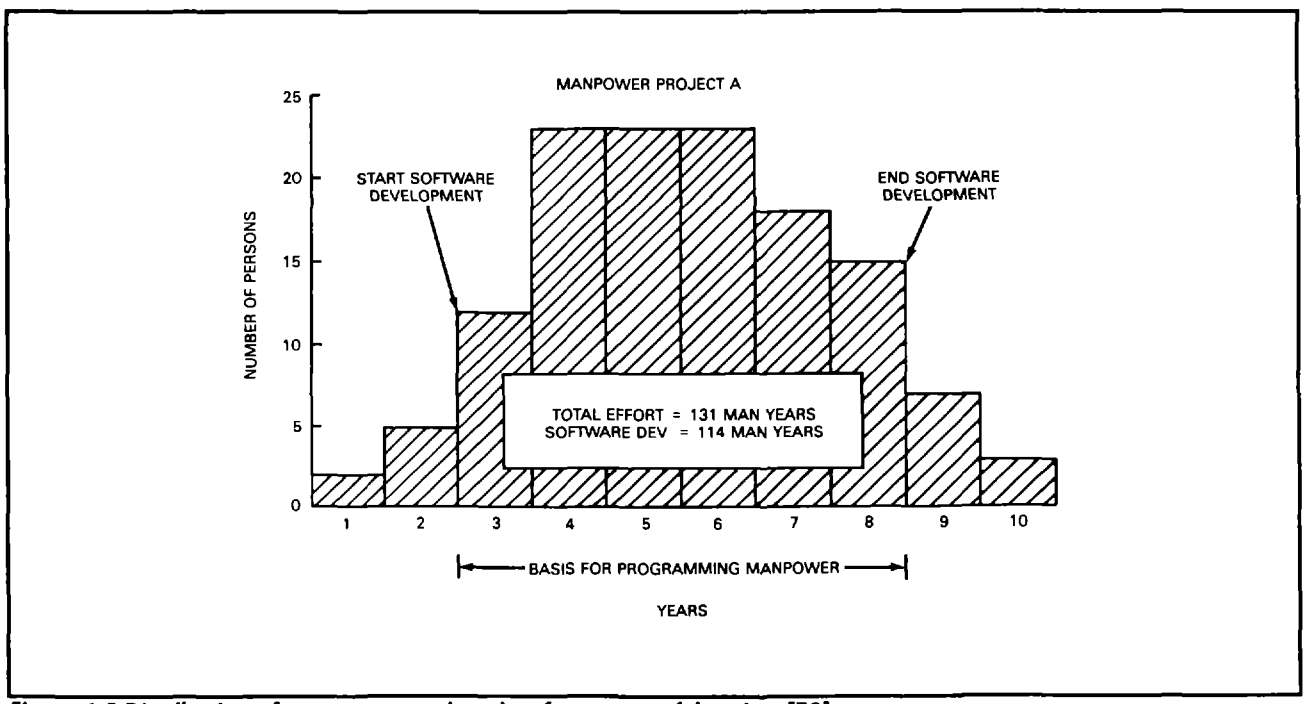

Figure 6-5 Distribution of manpower against time for a successful project [78].

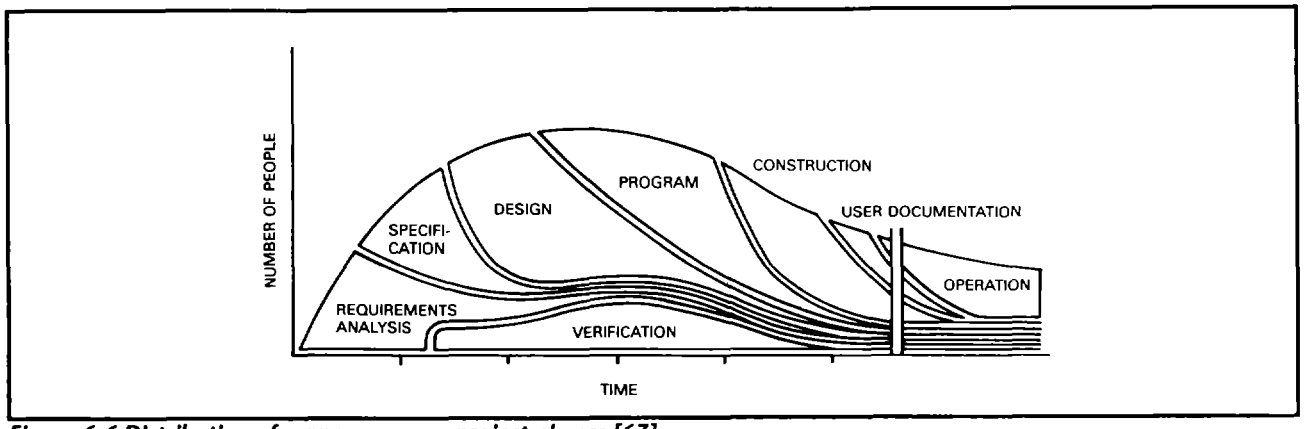

Figure 6-6 Distribution of manpower over project phases [67].

Program design:	26	(% of total project effort)
Coding:	24	
Testing:	36	
Documentation:	14	

However, two aspects should be emphasized here:

Firstly, it is important to know about the average values of such distributions, because one needs them for reliable estimates. The reasons for this are that reliable productivity figures can usually be obtained only for certain phases of the development cycle. One therefore has to extrapolate the total project costs from known figures for certain phases by using the statistical values for the other phases given in such distribution curves.

Secondly, the usual curves illustrate the maintenance problem. In whatever statistics one consults, maintenance costs usually amount to 50% of the total life cycle cost of the software. So, one has to be prepared to set aside a group of 10 people for the maintenance of a software system which cost 100 man-years to develop! Of course this situation is by no means acceptable and therefore every effort should be made to reduce the maintenance costs of the software by the use of better design method and good programming tools.

HUMAN FACTORS

General

This is one of the most important points a manager has to observe. All the planning and statistics will be utterly in vain if the manager does not succeed in keeping the team together and in maintaining a reasonable degree of job satisfaction and productivity.

There are many factors which influence this. The most important of these will be explained in the following paragraphs.

Motivation of the Team

To achieve motivation, the following factors were judged to be most important:

1. The team has to have a *fair chance of success.* That means that plans and schedules have to be feasible and realistic.

2. The individual team member has to have a *feeling of importance.* Never let the feeling arise that they are just regarded as a cogwheel which can be thrown away and replaced at any time.

3. The manager has to show an adequate *response to the needs* of the team. This means in the first place a proper working environment, but also includes the necessity to be able and willing to help people with their private problems as far as reasonably possible.

4. Always *maintain a slight overload.* This aspect was first emphasized by the Japanese, where it is generally accepted that people perform better and feel more satisfied if the manager makes them achieve a little more than they originally expected by themselves.

Team Building

1. The manager should perform a thorough *interest analysis* of the (future) team-members. In a profession like program development, which mainly depends on ideas and organization of thoughts, the performance of an individual obviously can vary by a factor of 10 - 20, depending on whether they are employed in the right place or not. And thus job satisfaction becomes an economically much more relevant factor than in many other more "traditional" professions.

2. Professional *ethics and morality* are more important than usually. Because complete testing and traditional quality control are not very well developed as far as software is concerned and even simply impossible in big systems, the commitment of the individual to do the very best job they can do, becomes an extremely important economic factor. This simply follows from the fact that a thoroughly developed program costs less in maintenance and in the damages caused by malfunctioning.

3. On the other hand, the manager has to maintain the *visibility* of the work of the team members in order to be able to properly perform control functions and to start corrective actions in time.

Dealing with Conflicts

1. Firstly, *identify and solve conflicts soon.* This would seem to be an old and well-known rule for team-leaders. But software people and managers generally have a predominantly technical background with little training in management and human factors, and therefore traditional rules of leadership are not very well known to them.

2. Secondly, *be prepared to create pain.* Technical conflicts can very rarely be resolved by a compromise and somebody has to lose.

3. But, also do not try to avoid conflicts at any price. *Conflicts are good for evolution* (this has long been discovered by philosophers) and, if handled properly, may even help those who lose one. They may win the next time.

Keeping Balance

One of the findings of human factors studies is illustrated in Figure 6-7. The manager has to be aware that humans are controlled by a field of tension in which they try to maintain a kind of equilibrium. It should be an interesting exercise for the reader to interpret this diagram for himself.

Another interesting observation is illustrated by Figure 6-8. There seems to be a correlation between the skill-level of team members and the number of meetings held. The consequences of this observation are not clear, because on the one hand meetings are good for communication, problem solving and conflict resolution, but on the other hand too much communication degrades productivity, as described in a previous section.

Special Properties of Software Teams

For decades a discussion has been going on among software professionals as to whether program development is a production activity like any other or whether it is something special to which traditional rules of management do not apply. However, it would seem that program development is truly comparable to traditional planning activities and that therefore software managers can learn a lot from other managers such as architects who plan large buildings, or from administration in civil service, railroads or military logistics.

One particular aspect of this problem can be stated as follows:

The majority of software professionals hold university degrees, although most are not in the field of software. This means that they have been educated into a tradition where they are judged for obtaining unique results. Usually university graduates also have never learned the necessity of the use of strict rules. Both backgrounds make it difficult to build sizable teams out of such people.

Of course the repetitive, deterministic part has always been much smaller in software projects than in more traditional construction projects. However to aid this situation in the future more emphasis should be given to the establishment of educational programs for a medium level of software people who are more trained in the direction of repetitive skills and the solution of small scale problems than their predecessors.

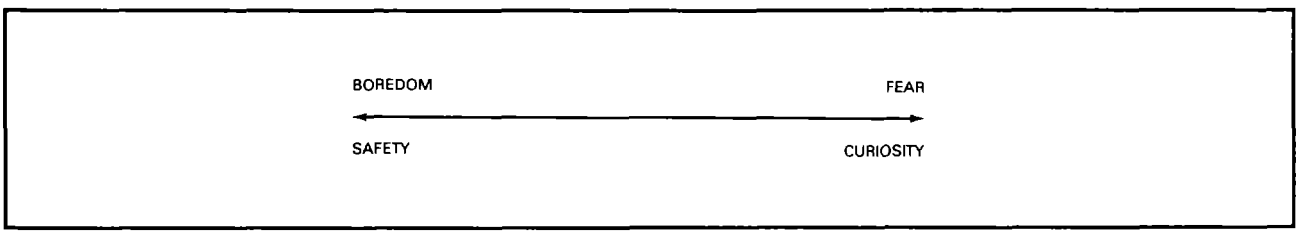

Figure 6-7 The psychological equilibrium.

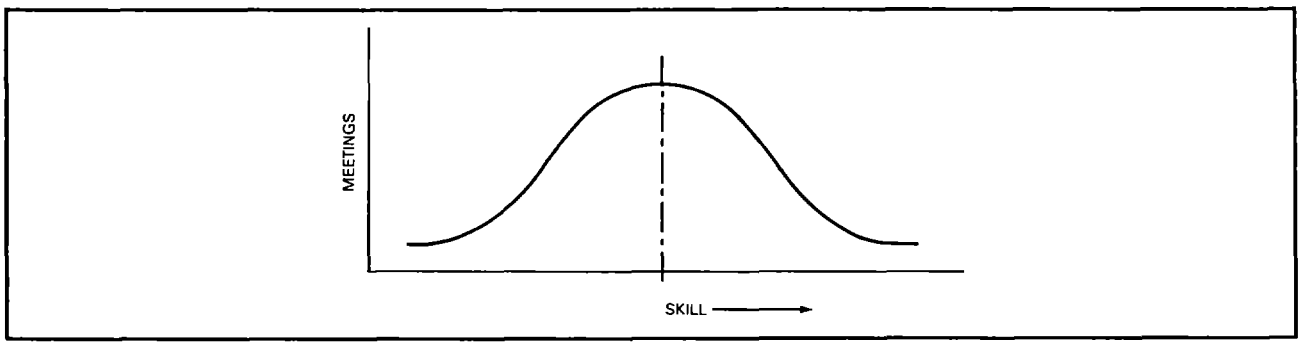

Figure 6-8 Dependence of number of meetings on skill level.

TECHNOLOGICAL ASPECTS

SELECTION OF SUPPORT TOOLS AND PROGRAMMING LANGUAGES

The following list comprises an incomplete list of the tools and languages which have been mentioned as having been successfully used. Intentionally it does not imply any order or ranking except an alphabetic one:

Ada,Ape, APL, AT-Xenix, BIGAM, BOIE, C, CA-DOS, CMS, COBOL, Codasyl, COMPASS, CORE, Dataflow-Diagrams, Debugging, Tools, DOS, EPOS, EXEC, FORTRAN, GMM, GESAL, ISP, JSD, LISP, MASCOT, MODULA2, Module-Management-System, Nassi-Shneiderman, PDL, PEARL, PET-MAESTRO, Petri-Nets, Pretty-Printer, Prolog, RCS, RMX86, RSX, RTE IV, SADT, SINET, SPA-DES, Structured Programming, Test Batch, Test Manager, TURBO-PASCAL, UNIPLEX, UNIX-Tools, VMS, Word Processors, X-tools, XEDIT, etc.

The overall list was compiled from a questionnaire distributed at a recent workshop [48] in which the participants were asked to mention all the tools and languages with which they had had experience and to indicate whether they had found them useful, neutral or counter-productive.

The evaluation of these questionnaires showed some interesting results.

1. 83 methods, tools or languages were mentioned, but only PASCAL, FORTRAN, UNIX, VMS, Structured Programming and Symbolic debuggers were listed more than twice in a positive sense.

2. 14 of them were criticized as counter productive and nine as having had no effect.

This means that it obviously does not matter very much which method or tool is used (if it is not too bad) as long as it is used professionally and in a consequent fashion. This view has been confirmed by several other studies.

It is more important for the success of a project that the team has experience with the support software, that it is readily available, stable and not too complicated to use. A major view of this, which has been formulated in [23] states: *The use of the tool should not require a higher intellectual effort than the solution of the problem at hand.*

It is also important to develop criteria by which software methods, tools and languages can be classified and judged with respect to their usefulness for any given project. Two first attempts in

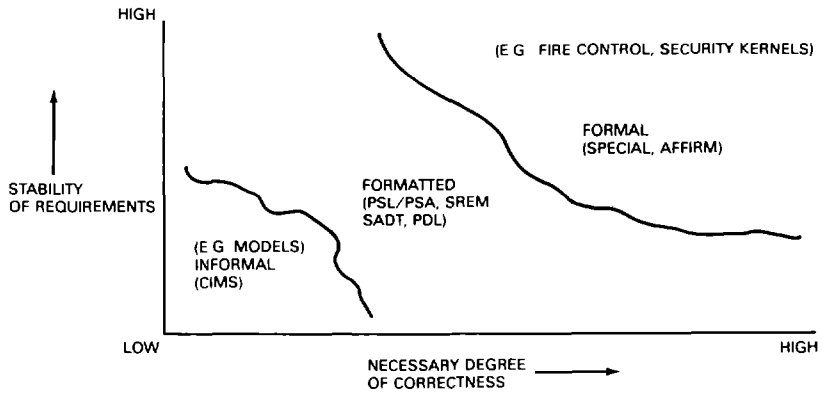

Figure 6-9 Regions of applicability of design tools [48].

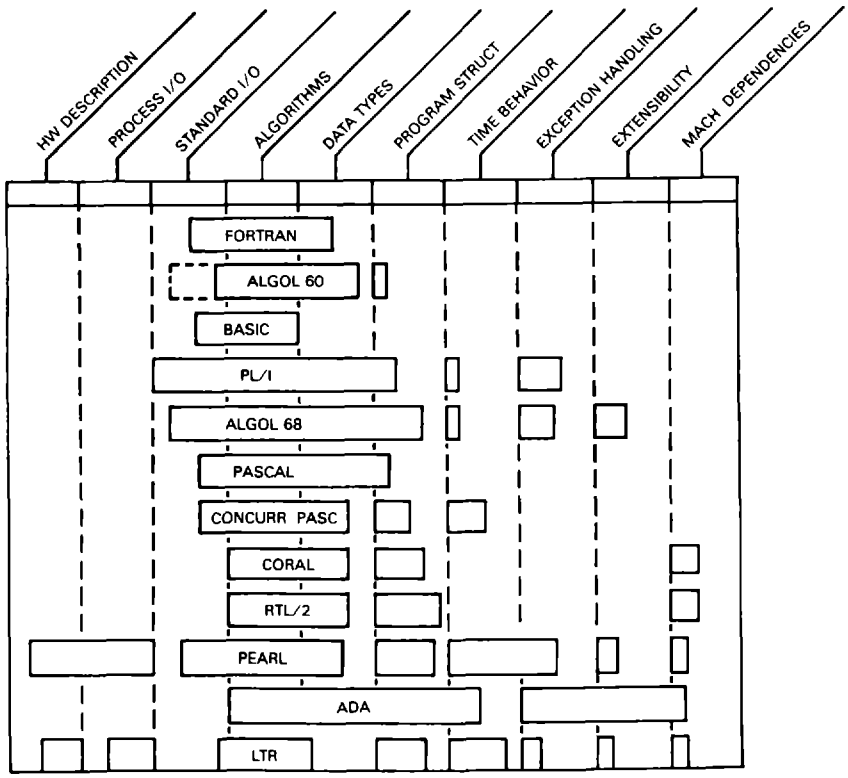

Figure 6-10 "Problem Coverage" by various programming languages [48].

ness for any given project. Two first attempts in this direction are illustrated by Figure 6-9 and Figure 6-10.

Figure 6-9 is adapted from [48] and shows regions of applicability for formal, formatted and informal software specification methods and tools.

The idea behind the scheme of Figure 6-10 is that programming languages can be regarded as formalized collections of those programming concepts which were well understood and therefore ripe for formalization at the time of the development of the respective language [48]. This view can be used by a software manager for the technical selection of the best programming language in the following way: identify the most important concepts in the application area of the project and select the language accordingly.

But in general it turns out that criteria such as the quality and stability of the compiler are economically much more important than many others for the usefulness of a particular language in a project. M. Key in [67] shows that in one particular project the forced use of an unproven language had caused an unnecessary expenditure of 200 man-years.

TEST TOOLS

The importance of test tools is in general grossly underestimated. On the one side there are not many useful tools for that purpose, on the other hand their use is almost never consciously planned. Both facts may of course be mutually interdependent. But the situation is so serious that it is necessary to break up this "vicious circle".

Figure 6-11, which is taken from [88], illustrates the reason: Experience shows that in most projects the detection of program bugs, i.e., the test coverage, follows the right curve. The explanation for this is obviously that people start out with a too optimistic view of the program error, or bug, rate in their program and test too lightly. Then, after major problems develop, they start testing in earnest and arrive at a program with 90% of the bugs out at "T90-real". If one would apply systematic testing from the beginning, one would obviously achieve a stable product much earlier ("T90-optimal") and thus save a lot of money.

The seriousness of the situation is further illustrated by the statistical evidence, that there are between 3 and 20 programming errors per 1000 lines of code before testing. Halstead [59] explains this by stating that there is a certain mental error rate which is different for each individual, but is rather constant for any one over time.

Of course it does not help much to state the seriousness of a problem if there is no solution for it. But in the case of testing there are promising methods and tools which are just not used widely enough. The available methods and tools can be roughly classified into informal and formal methods.

The *informal test methods* comprise:

1) Intuitive testing

2) Inspections

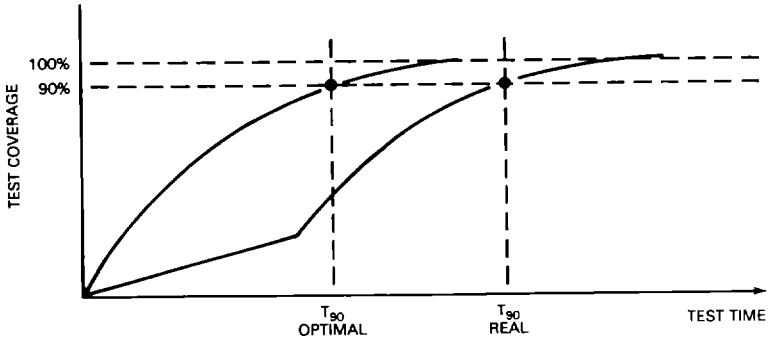

Figure 6-11 Test coverage over time.

3) "Walk throughs"

4) Test plans

5) Program controlled testing

6) Limit testing

7) Special test environments

The *formal test methods* comprise mainly what is other-wise known as *program verification*:

1) Analytic/logic verification ("program proofs")

2) Program-flow oriented verification

3) Data-flow oriented verification

For a number of languages there exist verification tools, for example, for:

FORTRAN:	ATTEST, DAVE, DISSECT, FACES, FAST, PET, RXVP, SADAT, SQLAB
PASCAL:	RXVP, SQLAB
JOVIAL:	JAVS, RXVP
PEARL:	PEARL-Analyzer

But one warning should also be given here: If used uncritically, many of these methods result in the production of enormous quantities of paper which in turn are very difficult to evaluate. Thus, a programming team should gain experience with them in pilot studies before using them in a full-sized project.

DEVELOPMENT SUPPORT HARDWARE

This is another problem area which is often not dealt with in relation to its true importance. Sufficient development support hardware is an important productivity factor and one usually needs more than is available. But it is important for the manager to know this in advance and to plan for the necessary funds in order to provide it at the right time. Figure 6-12 shows a typical curve for the support hardware needed during a major project. It has been taken from [67].

FUTURE TRENDS

It was generally agreed that the technological situation in the field of software development had improved over the past ten years and that at present there are enough tools available. The main problem today is how to use them properly. But the coverage of the software development cycle by tools is still very inhomogeneous and in some areas further developments are necessary. The following potential future developments have been identified as necessary and feasible:

1) "Intelligent" tools

2) "Contents" - or "concept"-oriented programming

3) Graphic user interface

4) Built-in-simulation

5) Integration of "rapid prototyping"

6) Language independence

7) Machine independence

8) Automatic generation of test data

9) Automatic generation of error handlers

10) Management visibility

11) Generics and macro processors

12) Guidelines for software and system development

In order to determine the priorities for these goals, the participants of the recent workshop [48] were asked to rank the proposals under two different boundary conditions:

A) Regardless of cost and only according to their technical merits and necessity.

B) With major consideration of project cost, i.e., if people had to pay for the development themselves.

This has resulted in the following order of importance in each case:

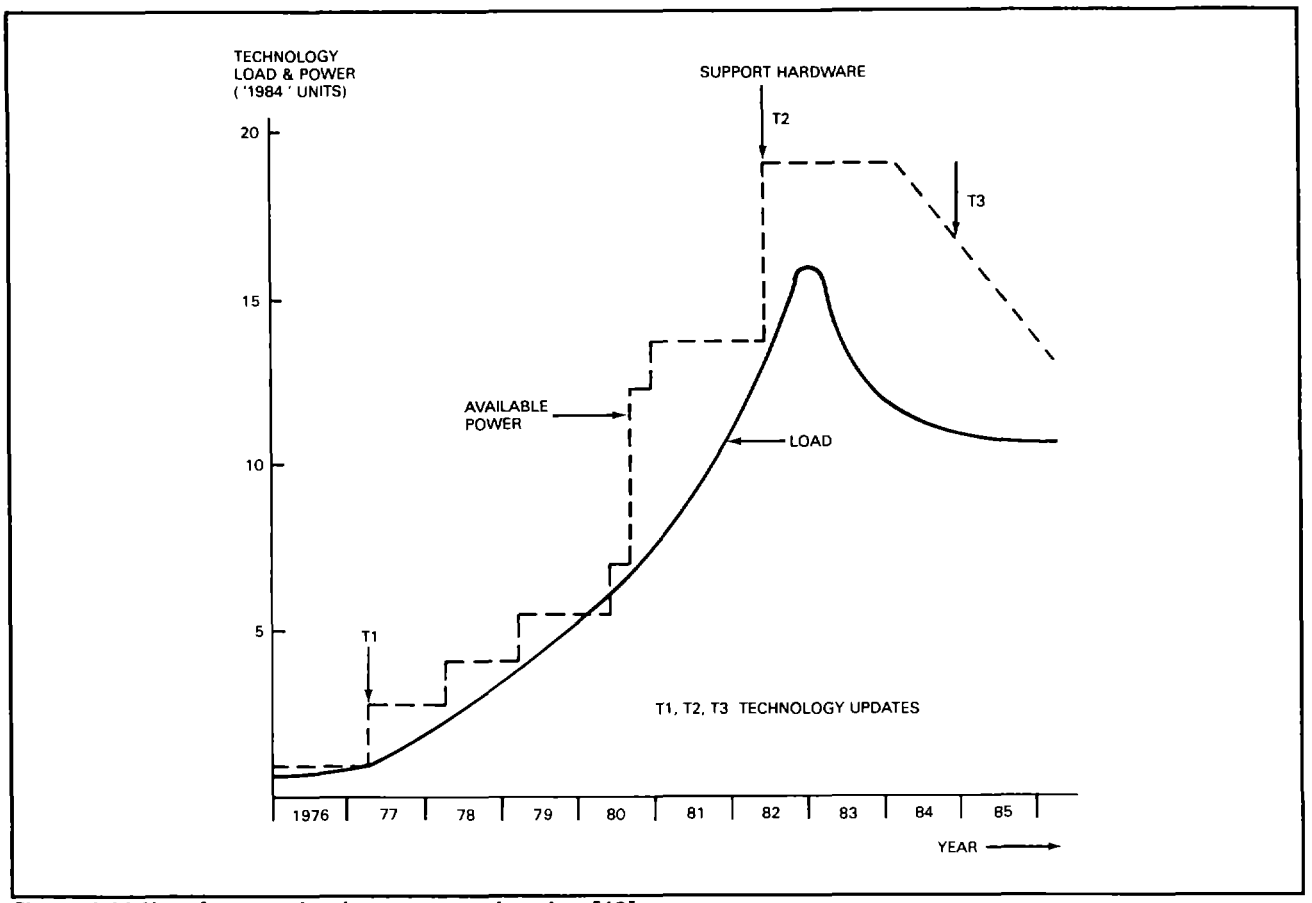

Figure 6-12 Use of support hardware over project time [48].

A) 1. "Intelligent" tools

2. Guidelines
Integration of rapid prototyping

3. Graphic user interface
Machine independence
Management visibility.

B) 1. Graphic user interface

2. Integration of rapid prototyping

3. Guidelines

4. Machine independence
Management visibility
Concept-oriented programming

SUMMARY

The current state of the "Art of Software Management" can be summarized as follows:

1. Technically there are still problems but there are enough methods and tools around to properly support a project.

2. It is necessary to train managers better in order to enable them to:

   a. Properly use all these tools

   b. Organize their teams

   c. Motivate their people

   d. Control their resources

Management of software projects has to and can be learned and should not just be based on technological beliefs.

BLOCK DIAGRAMS OF THE PROGRAMMING REQUIREMENTS FOR THE SCHEDULING AND CONTROL HIERARCHY PROGRAM MODULARITY

As this generalized reference model readily shows, program modularity is the key to future transportability and reuse of computer programs in succeeding integrated production plant computer control systems. Modules themselves must be organized into sub-modules such that all possible commonality between comparable programs is preserved in the overall structure of the program and differences are concentrated in replaceable sub-modules which are specific to the particular applications involved. That is, program modules must be made as generic as practicable.

This is obviously not a new thought with the Committee and in fact is a well-known software engineering technique. The problem which exists is one of coordinating the design of these program modules so that their interfaces with other modules to which they interconnect are minimized. In addition, the modularized programs must themselves be written in a language which has been standardized for the real-time applications needed here.

AN EXAMPLE MODULAR PROGRAMMING SYSTEM

The first Figure (6-13) of this Section presents a diagram of the operations which are executed by programming in the process computer system. This diagram shows the overall system as carried out by a single computer containing all functions. The following figures show the corresponding diagrams for each level of a hierarchy computer system in turn. These diagrams correspond to the

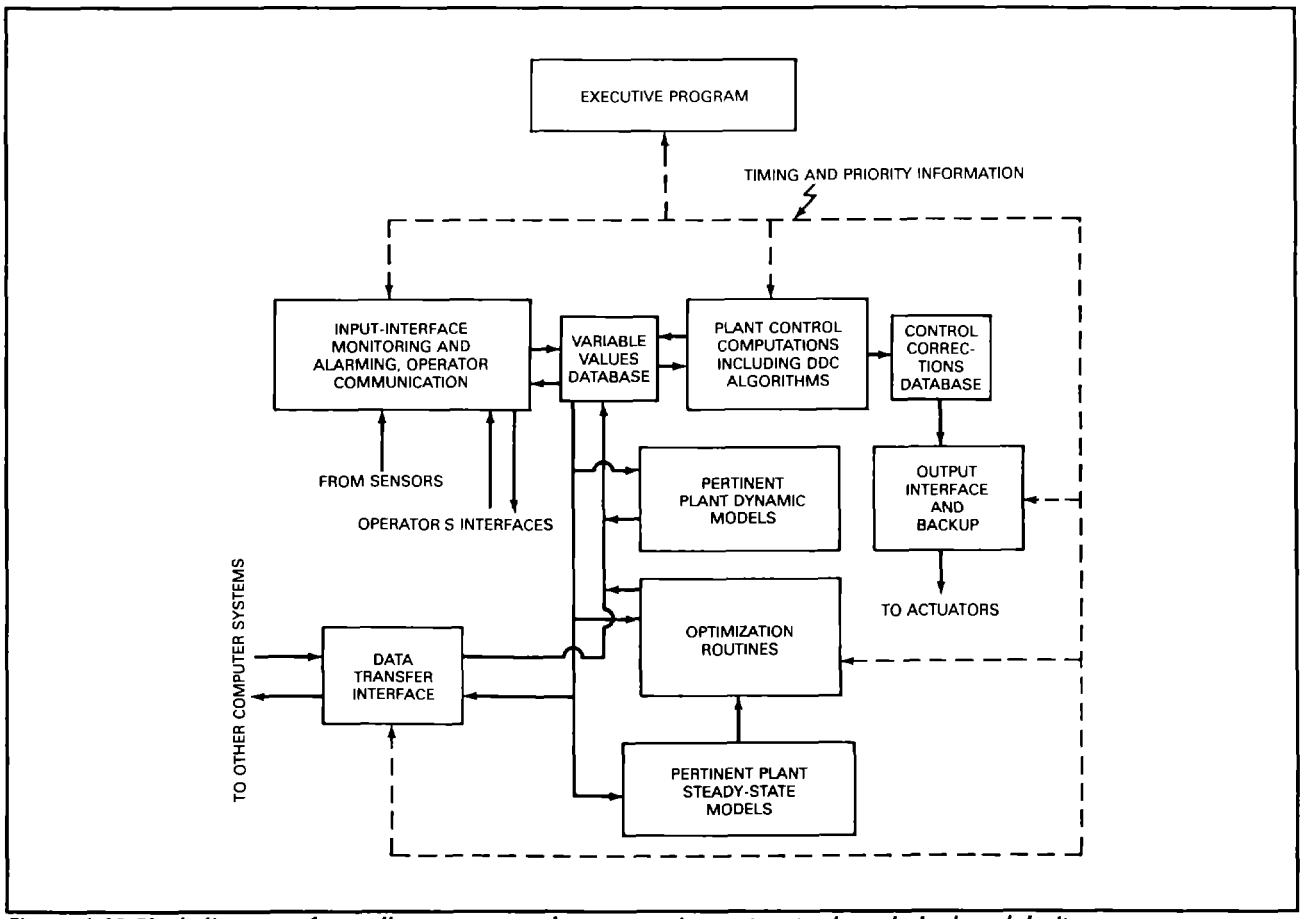

Figure 6-13 Block diagram of overall process control programming system to show desired modularity.

figures of the earlier Chapter in showing the duties carried out at each level and corresponding software modification.

Such a modular system allows any particular modules to be modified without affecting any of the other modules, thus greatly simplifying both the initial programming effort and any later required program modifications. This is made possible by the use of the database elements indicated in the diagrams. A further advantage of such a program is the fact that programs developed by others for any of the modules can be readily integrated into the overall program. The chance of finding a suitable preprogrammed module is obviously much more likely than the corresponding chance of finding the complete overall program for any particular specific application.

It should be noted that most of the differences between process control programming functions and engineering or business type programs are included in Level 1 of the hierarchy system, the second figure. Thus, the higher-level functions can probably use many programs developed for other applications. This probably will not be true for the needs for programs for the supervisors' and managers' interfaces or for any remaining time-based functions since these are the functions

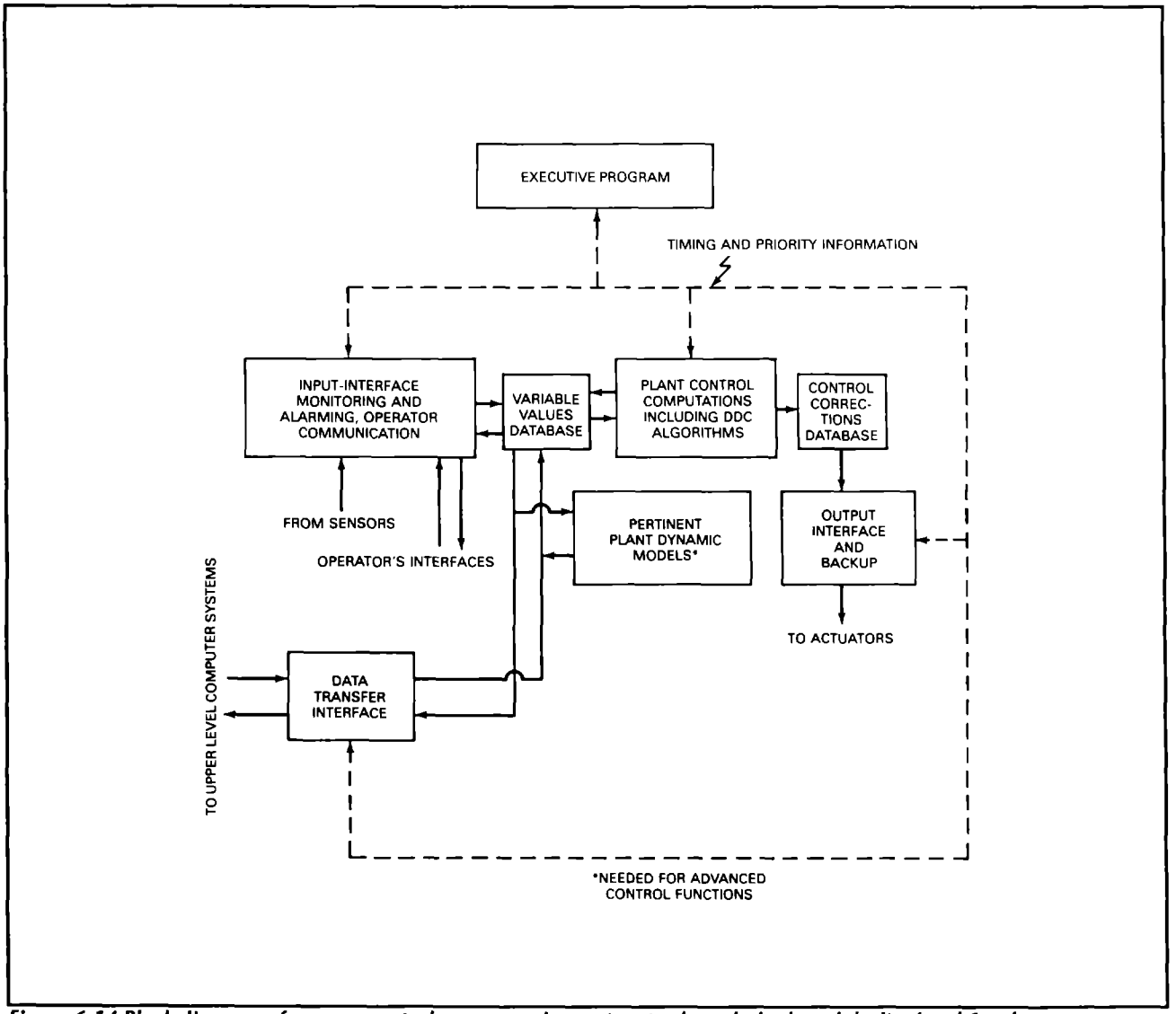

Figure 6-14 Block diagram of process control programming system to show desired modularity, Level 1 only.

which are least likely to be used for the business and scientific application fields.

While the diagrams of Figures 6-13 to 6-17 have been drawn for the main line process control, production control and production management tasks of the plant, it can readily be seen that the diagrams would apply equally well to the auxiliary tasks necessary in plant operation. These are maintenance management, raw material and energy control, product inventory control and statistical process control, among others. It can be readily seen that these functions would need access to the appropriate databases, would communicate both up and down in the hierarchy and would have the required man/machine interfaces. They would carry out the necessary task computations using the associated standard algorithms and related plant models. Most of these functions could use the standard process control sensors for any needed plant data. Thus they would probably not need any large number of special sensors.

The work involved would generally take place at levels higher than Level 1, probably at Level 2 or Level 3. Thus Figures 6-15 and 6-16, appropriately modified to include the terms common with the auxiliary tasks mentioned above, would readily apply to diagram these functions.

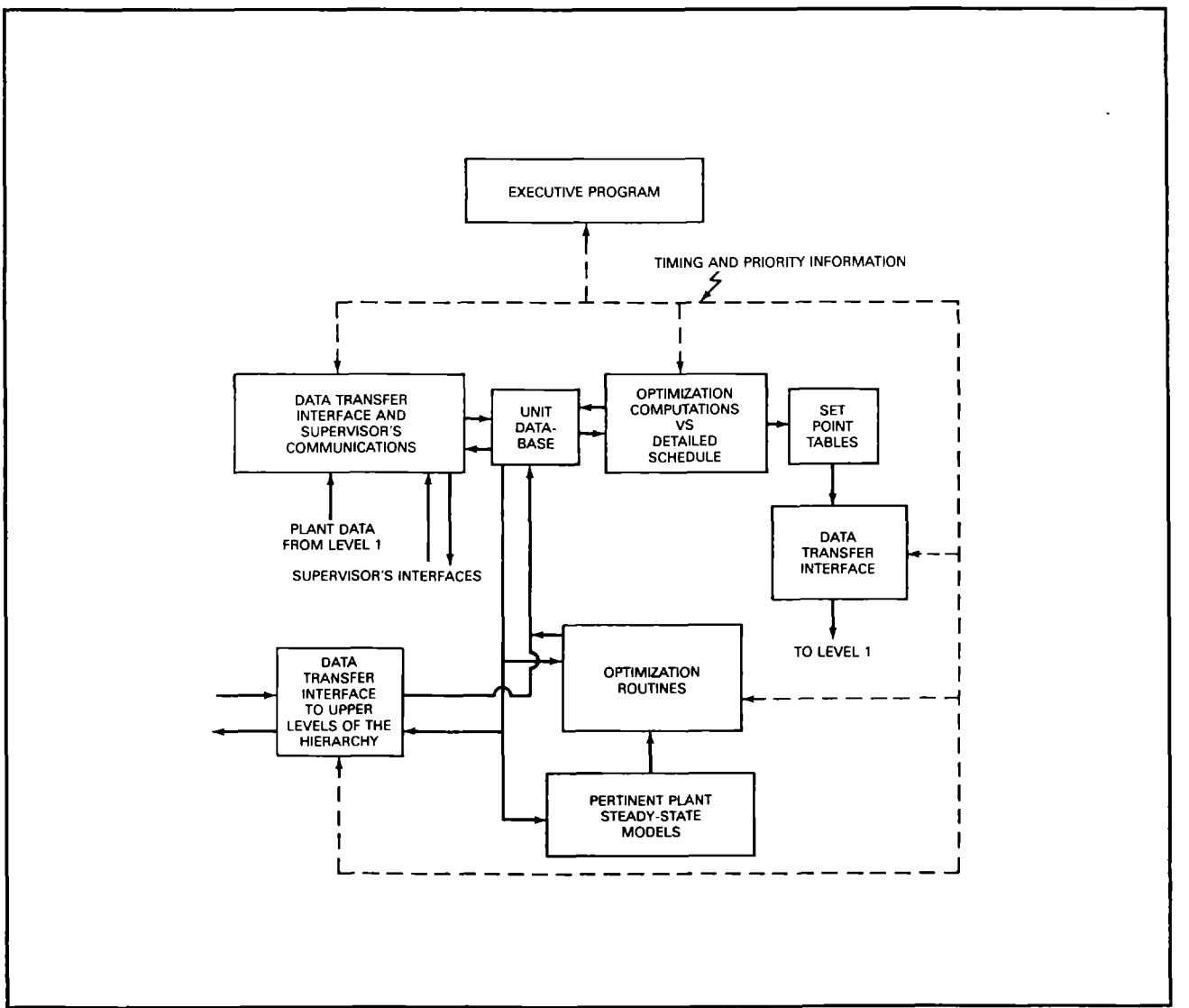

Figure 6-15 Block diagram of optimizing control programming system to show desired modularity, Level 2 only.

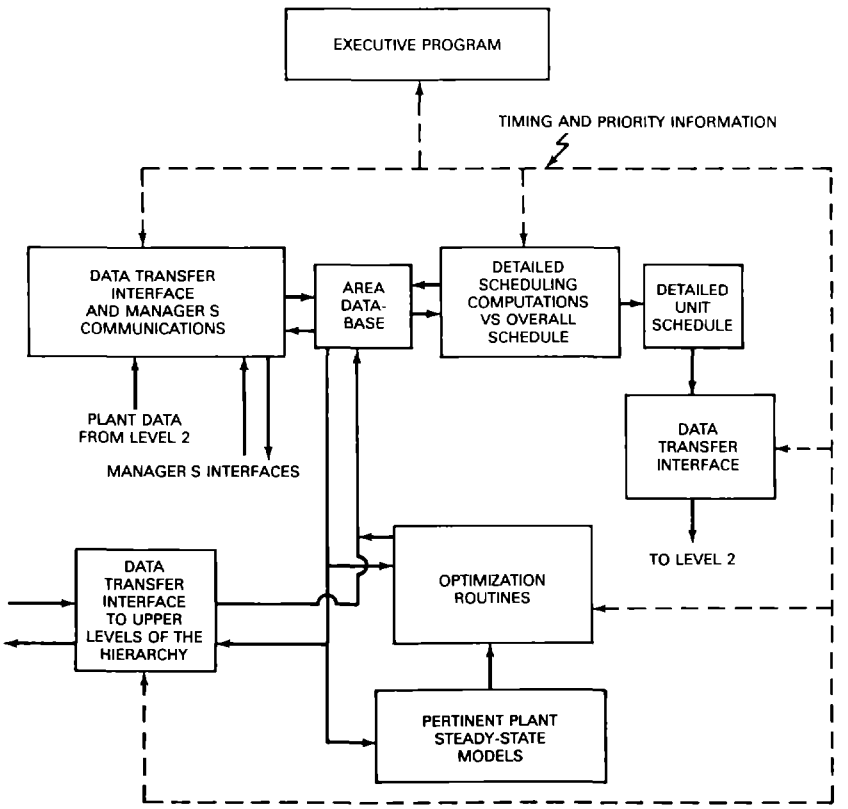

Figure 6-16 Block diagram of detailed scheduling programming system to show desired modularity, Level 3 only.

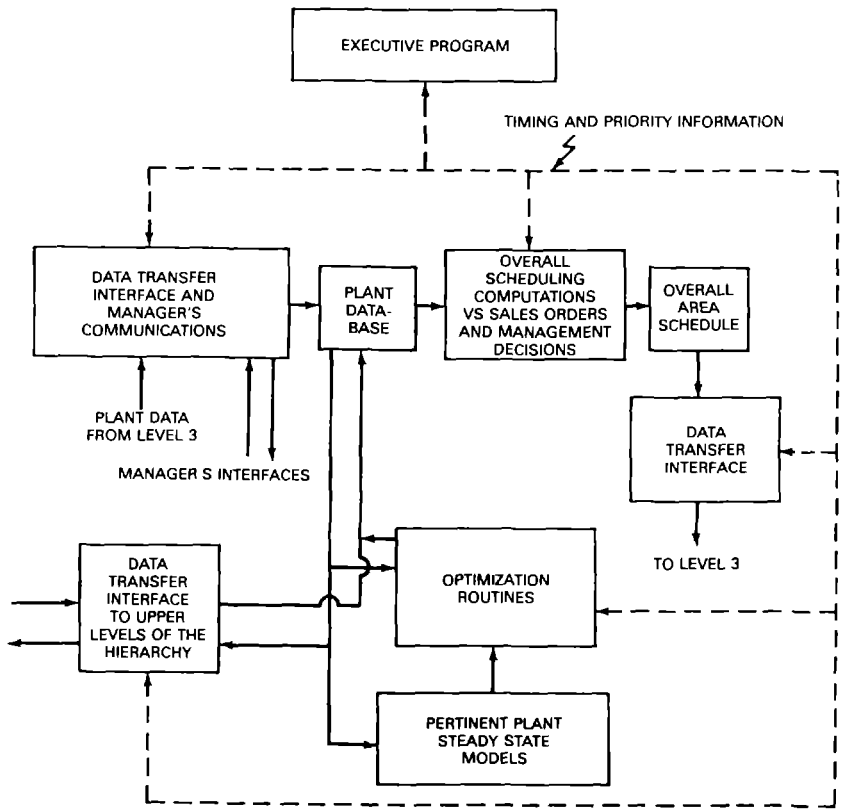

Figure 6-17 Block diagram of overall scheduling programming system to show desired modularity, Level 4 only.