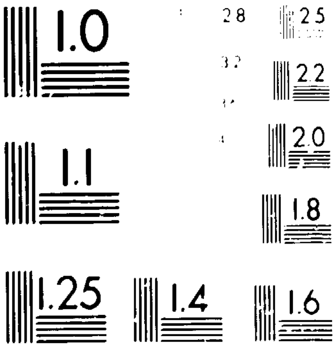

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

13239

Distr.
LIMITED

UNIDO/IS.440 ←
10 February 1984

ENGLISH

UNITED NATIONS
INDUSTRIAL DEVELOPMENT ORGANIZATION

GUIDELINES FOR SOFTWARE PRODUCTION IN DEVELOPING COUNTRIES*

by

Hermann Kopetz**

* The views expressed in this paper are those of the author and do not necessarily reflect the views of the secretariat of UNIDO. Mention of firm names and commercial products does not imply the endorsement of UNIDO.
This document has been reproduced without formal editing.

** Professor, Technische Universitaet, Vienna, Austria

V.84-81528

Guidelines for Software Production in Developing Countries

+ UNIX is Trademark of Bell Laboratories

++ ADA is a Trademark of the US Department of Defense

# SADT is a Trademark of SOFTECH

Terms which are marked with an "*" are defined in the glossary at the end of these guidelines.

1. INTRODUCTION

In the last thirty years, the information industry has become one of the major industries in the industrialized nations. It is the main growth industry and the driving force for productivity gains in many other fields in the economy. The "automation of information processing" provides tremendous opportunities. Significant improvements in many areas can be realized, e.g.

- decision making
- production
- education
- environmental monitoring
- medicine
- communication

etc..

But there has also been a considerable impact of the information industry on the less developed countries, both direct and indirect.

The direct effect of the information industry is the application of computers in these countries in order to assist in the handling and processing of information. Such applications range from commercial data processing to scientific calculations, production control, process control etc.. However in countries, which do not have an adequate infrastructure for the operation of computers (i.e software* development and maintenance) the installed hardware base is of relatively little use.

The indirect effects have not all been positive. E.g. the application of computers in the automation of assembly line processes in the industrialized countries has led to a considerable reduction in the prices for these goods, since the amount of human labor has been reduced. This does not only lead to some employment problems in the developed countries, but also to some production problems in developing countries, since it is difficult to compete with these automatically manufactured goods on the world markets.

Generally speaking, some of the negative consequences of the "information age" will thus confront many societies, particularly those which do not actively address this new phenomenon. The positive aspects of the "information revolution" can only materialize if an active policy in respect to information processing is formulated and executed.

It is the aim of the following report to introduce some of the basic concepts of the information industry, particularly the software* side, and to provide some Guidelines for software production in the Developing Countries.

1.1 What is "software*"?

The main motivation for the first commercially available computers was the need to assist the engineer in the solution of rather simple numerical problems. In these application, the problem structure and the solution algorithms* are well specified.

With a conventional calculator it is necessary to present each step of the solution to the machine immediately before it is executed. With a stored program* computer, one can specify the necessary steps of the computation i.e. the algorithm* in advance and store them in the machine. It is then possible for the machine to execute all the steps of the computation autonomously. We call the specification of the solution algorithm* in a machine readable form a "program*". The notation, which is used for this specification* is called the "programming language*". Programming is thus concerned with the translation of a given algorithm* into a form, which can be stored and interpreted by a computer.

The physical units of the computer, e.g. the processing element, the storage device, and Input/Output device etc., are called the computer hardware, in short "Hardware*".

In order to make use of the Hardware, a set of programs and a procedure with associated documentation, which tells the user how to operate the programs, is needed.

The set of all computer programs, procedures and the associated documentation pertaining to the operation of the computer, is called the computer software, in short "Software*".

In the beginning of the computer era, the main difficulties were concerned with the design* and implementation of the physical machine, i.e. the hardware. Compared with this enormous effort the development of the -- at that time -- small and well specified programs was relatively simple.

As time progressed, the problem of building reliable hardware was attacked by many institutions and tremendous progress has been made in this field in the last thirty years.
In the mean time, computers have been applied to the solution of large and complex problems, which are not as well specified as simple engineering calculations. The task of specifying and implementing the required software* has grown considerably. Since about 1968, the problems associated with the specification and implementation of large software* systems has been recognised as a major challenge to the computer profession. A new field, "Software Engineering" has emerged as a discipline of its own right.

A number of important developments has taken place in the software* field since the first days of the computer. New high level programming languages, which are more easily understandable to the human programmer, have ousted the low level languages, which are implemented by the hardware* engineer. Special programs, the compilers, have been developed which trarslate the high level language* representation of a program to the required machine language* representation automatically.

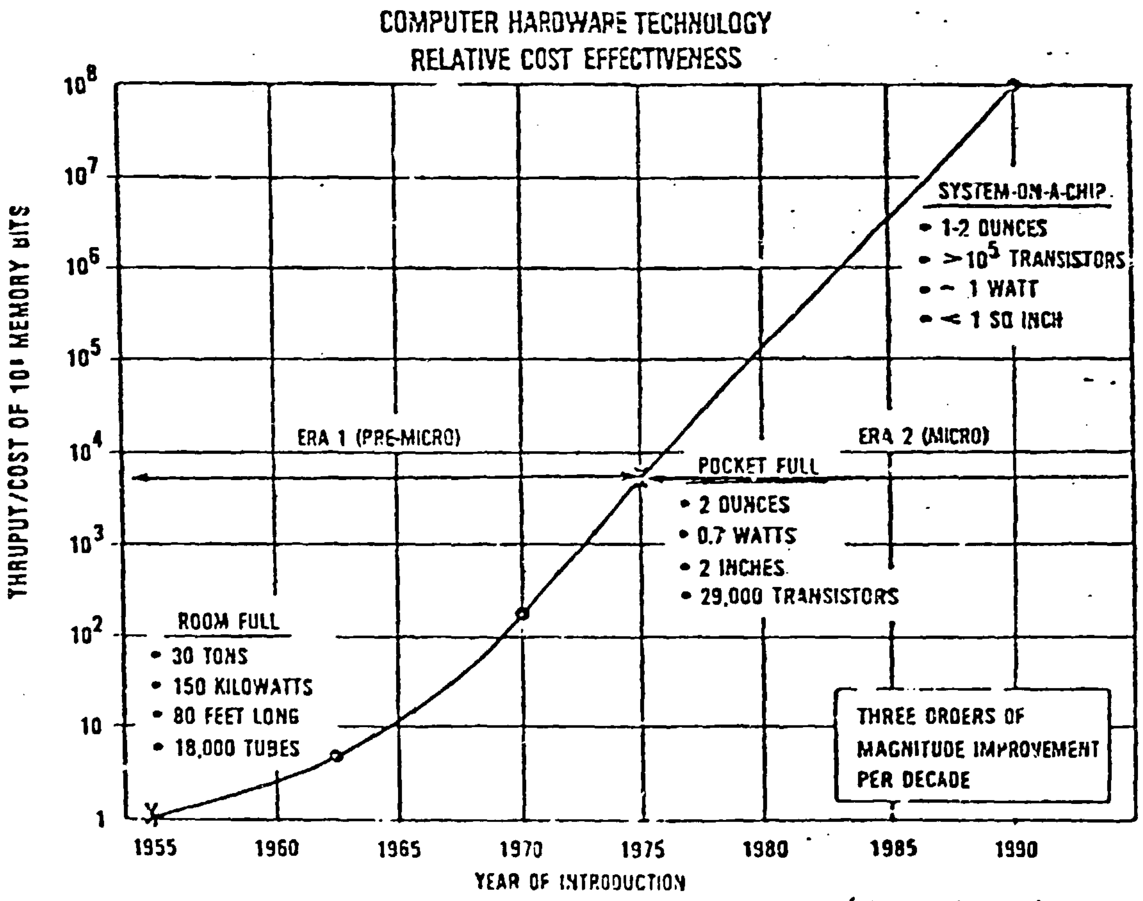
It has been found out that the management of large software* projects requires special attention. Therefore new techniques for the management of large software* projects have been developed.

However, taken all aspects together, the software* field has not progressed as fast as the hardware* field. The main obstacles for a wider and more beneficial application of computers are in the software* field.

1.2 The role of software* and hardware* in the Information industry

In the early years of computing, the computer industry was primarily based on the large central machines, the mainframes, whose cost were generally well in excess of US $ 100.000 a unit. The interaction with these mainframe computers was limited to skilled professionals.

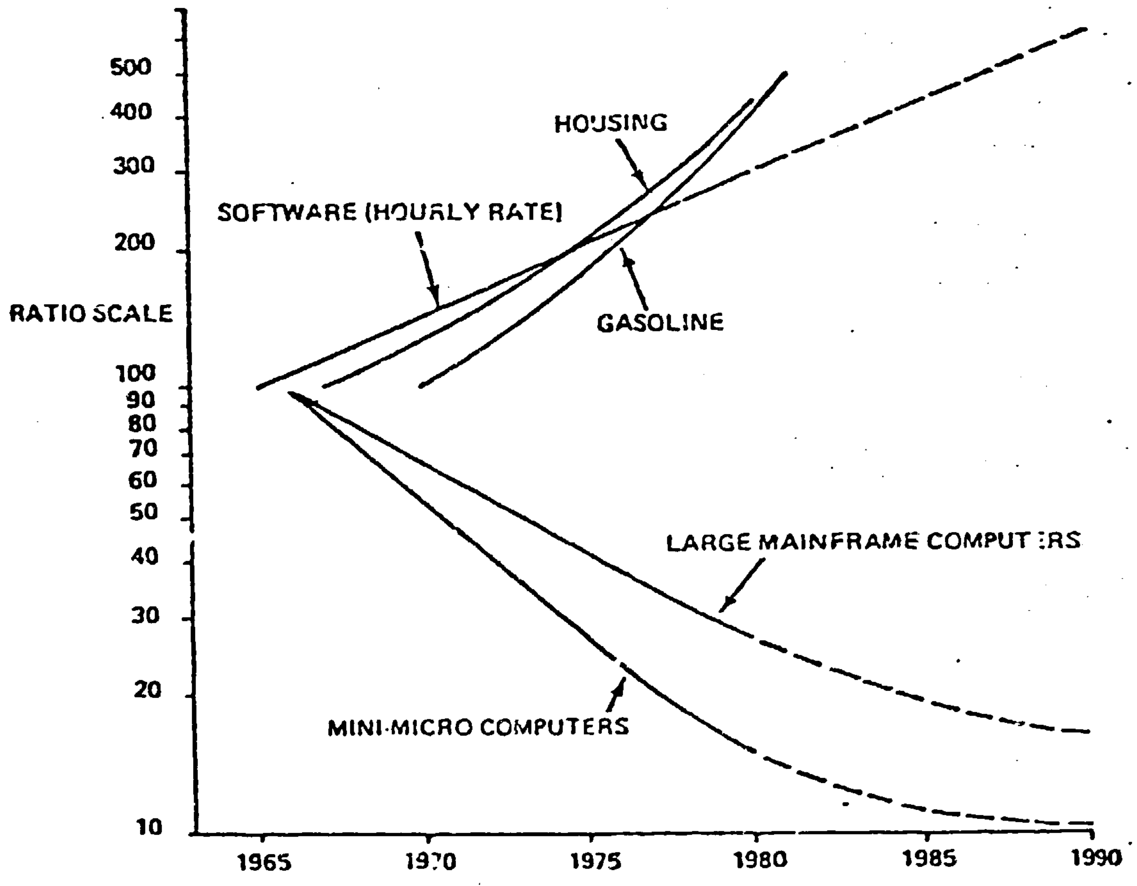
With the advent of the microelectronic revolution, the prices of the computer hardware dropped sharply. The following diagram shows the increase of the cost-effectiveness of computer hardware* in the last twenty years.



/Musa,1983/ p. 6

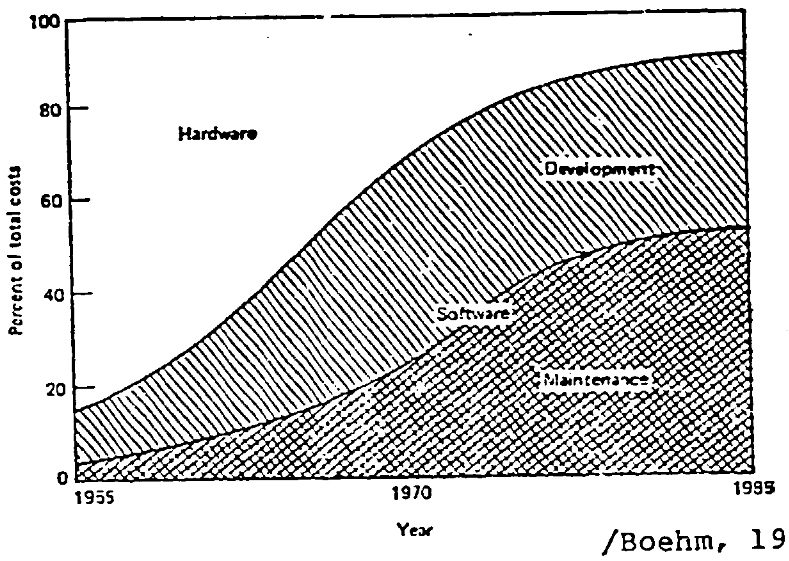
Powerful systems for commercial, scientific and engineering applications can now be acquired for less than US S 10.000 The personal computer market, which has only been around for about five years, is burgeoning. Machines in the price range from a few hundred US $ upwards make the computer to a product for the small business and even for the end consumer. These developments have increased the general involvement of the public with computers and software*.

Compared to this tremendous increase of three orders of magnitude per decade in the cost effectiveness of computer hardware*, the productivity gain in the field of software* was rather moderate, as is shown in the following diagram.



/Musa, 1983/ p. 8

As a consequence of these developments, the ratio of software* costs to hardware* costs in computer projects has changed considerably over the past thirty years.



/Boehm, 1981/ p.18

About 90% of the effort, which goes into the design, implementation and maintenance of a computer project, is in the area of software* (including management). The hardware* part, that is the physical equipment, amounts to only 10 %. At this point it is important to remember that software* is basically an intellectual product. <u>The investment needed to create a software* industry is mainly in the area of education and not in substantial capital equipment.</u>

This is one of the reasons why there are challenging opportunities for many countries in the area of software* engineering. The application software* has to incorporate the specific legal and organizational rules of a society. An accounting package, for example, must be tailored to the requirements of the given legal system.
These local requirements can form the starting point for a software* industry, which later on can be expanded to cover also standard packages of wider applicability.

It can be expected, that the information industry will have a similar impact on the economy of less developed countries as on the economy of highly industrialized countries, although with a certain time delay. It is therefore reasonable to look to the highly industrialized countries to get an indication for the potential impact and future of this industry.

In a recent report about software* engineering progress which has been published by some of the best known experts in the field, the following has been said about the future of the computer industry /Musa 1983,p7-8/:

"We observe that computers are becoming smaller and cheaper, and that they are being distributed to a wider and wider population. Important current trends include:

(1) Decreasing hardware* costs.

(2) Increasing share of computing costs attributable to software*.

(3) Increasing range of applications, to the extent that the dependence of society on computers is becoming more and more critical.

(4) Continuing or increasing shortage of qualified software* professionals.

(5) Continuing lack of appreciation of the nature of software* (it's not actually "soft", it's rarely capitalized, it's difficult to evaluate quantitatively).

(6) Increasing development of distributed computing and convenient network access.

(7) Increasing availability of computing power, especially in homes.

(8) A widening view of computers as an information utility; anticipation of the "automated office".

(9) Increasing quality of interfaces to humans (voice, high speed and high resolution graphics).

(10) Increasing exposure of nonprofessional people to computers.

On the basis of these trends, we can extrapolate some future developments:

(1) Pervasive Consumer Computing
Computers will be extremely wide spread, both as multiple purpose machines in homes and offices and as dedicated (embedded) machines for applications such as household environment control. Most of the users of these machines will be naive--certainly the majority of them will not be programmers

(2) Information Utility: We will come to think of computers primarily as tools for accessing information, rather than primarily as calculating machines. Networks will provide a medium for making available numerous public data* bases, both passive (catalogs, library facilities, newspapers) and active (newsletters, individualized entertainment). Distributed applications such as electronic funds transfer will become common. Electronic mail will reach a substantial fraction of the population.

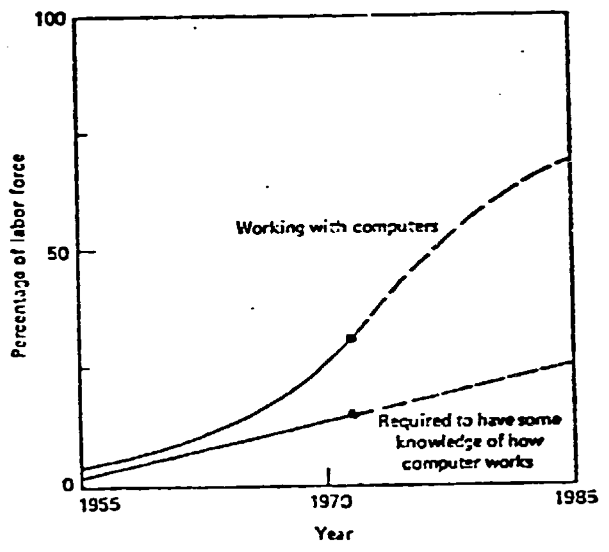
(3) Broad range of applications: The range of applications will continue to broaden, and almost all areas of society will be critically dependent upon computers. As a result of this pervasiveness and criticality and widespread use by nonprogrammers, much of the software* will provide packaged services that require little, if any, programming. The packages will be tailored to individual needs, but not necessarily by individual users. Turnkey systems will become even more common in the business world, and there will be substantial economic incentives for producing general systems that can be applied to individual, possibly idiosynscratic, requirements.

(4) Changes in the Work Place: Distributed systems and networks will facilitate a distributed work place, but we doubt that the norm for the office workers will be to work at home instead of in an office--computers will not replace human interaction for decision making. The potential for software* development as a cottage industry will increase. Electronic work stations will change the nature of work that now depends on paper flow, and robotics will substantially change manufacturing.

(5) Changes in Education: We can already see the effects of pocket calculators and personal computers on the teaching of mathematics and many other subjects and on students' expectations about the educational process."

It has been estimated, that already nowadays more than 50 % of the work force of some highly industrialized countries deals in one form or another with information management. The potential market of the computer industry, and particularly the software* industry, is thus considerable.

The following diagram shows the percentage of the US working force which will rely in one way or another on computers and software*.



/Boehm, 1981/, p.19

The computer industry is thus the key to modern technology. since many high technology products are based or depend on the application of computers. Technology transfer without a high level of computer literacy will be more and more difficult.

With respect to the overall computer and information processing industry of the future, computer software* will be the dominant portion of an industry, which will grow to more than 10 % of the GNP of the United States by 1990.

2. THE NEED FOR A SOFTWARE POLICY IN DEVELOPING COUNTRIES

As already mentioned before, the impact of the information indust ʼ on the developing countries is significant. In order to realize the maximum benefit for the society, an active information policy should be formulated and executed in every country. This policy should try to take advantage of the positive aspects of this new field and to avoid, or at least reduce, some of their negative consequences .

Some of the positive aspects of the information industry, in particularily the software* industry, as seen from developing countries, are:

- The functionality and user interface of a computer is determined by its software*. A local software industry can thus produce computer systems, which are well adapted to the needs of the given society.

- With adequate software*, computers can provide valuable assistance in areas like general education, vocational training, operation and maintenance of industrial equipment, just to name a few.

- The development of software*, which is a work intensive proce:s, accounts for the major section of the new information processing industry.

- Software production requires relatively little capital equipment. The main requirements are interested people with good training and guidance and access to computer hardware.

If no active information policy is persued, the following negative consequences can result:

- Many beneficial applications of computers are not realized, because there is little understanding about the capabilities and potential of computers.

- Local job opportunities in the software* field are missing and some of the foreign currency is spent on software* products, which --just as well--could be manufactured locally.

- The society becomes dependent on software*, which is supplied from outside. This software* is not well adapted to the needs of the local users.

An active software* policy is to address the following areas:

- organisation
- training
- standards

The development of policy guidelines to cover these topics will be the major concern of the next sections.

2.1 Organization

The startup of any local software* industry has to be based on the development of application software*. A substantial portion of the application software* is specific to the local environment and is thus a "protected" market.

If we analyse the world-wide market for application software*, we will find out that this market has some characteristics of a "cottage industry". Small companies and sometimes even single consultants play an important role in this market. If the startups in the area of application software* are to be successful, it is necessary to provide a climate in which such a "cottage industry" can blossom.

It is the responsibility of the political decisionmaker to provide the organizational framework for such a climate. This can be done by some organizational unit -- we will call it "office for information technology". This office of information technology should provide the following services:

- Monitor the market of information industry products and new trends and developments. This will provide valuable background information, both for the political decision makers and software* developers.

- Formulate and, after approval, execute an initiative to promote general "computer literacy". A necessary prerequisite for the success of a new industry, such as software*, is a general public awareness for the potential, the capabilities and the risks involved. This is predominantly an educational endeavor. A more detailed outline of such a "computer literacy" inititative will be given in the following section under the heading "training".

- Initiate a research program* on information science and specifically on software* technology. This is also an educational endeavor and will be discussed in the following section.

- Support interested individuals and startup companies in the area of software* technology with economic and legal advise, as well as with some financial assistance.

- Set up the legal framework for the introduction and execution of standards for the information industry. Standardization of hardware, software* and the associated documentation can substantially reduce the maintenance costs for information industry products and improve the compatibility of the different systems. Standards will be discussed in more detail in a subsequent section.

The office of information industry should not be a big bureaucratic organization. It should be a small group of highly competent experts which is responsive to their clients.

2.2 The Establishment of Training Facilities

The most important result of an active Information Policy is the general advancement of computer literacy in a society. A high level of computer literacy is a solid base for good computer applications, a successful software* industry and the critical appreciation for the benefits and risks of this new technology. Since many high technology products are based on computers, a high level of computer literacy provides also a positive climate for technology transfer in general.

It must be the goal of a computer literacy initiative to bring a high percentage of the youth in direct contact with computers and software* at an early age. Personal computers should be installed in all schools and students in the age of ten upwards should have the possibility to use these machines in their mathematics and science classes. Experience has shown that students of that age have no problem in mastering the computer. If there is some lack of trained teachers, computer assisted instruction courses, which run on these small machines, can fill part of the gap. Students, which thus develop a natural relationship with the computer, will have no difficulty in integrating the computer into their workplace at a later stage.

While it is important to introduce computers into the general education system as early as possible, the retraining of parts of the active workforce must not be overlooked. Many job profiles are changed because of the introduction of computers. It is irresponsible to fill new job positions with new people only and to push those workers, who do not have the necessary knowledge in the new technologies, aside. The persons, who have worked in a given position for a number of years, have gained valuable job experience which, combined with some software* knowledge, is an important asset to society. The computer and software* training of the adult population must thus be included in the training programs.

The training of teachers and software* experts can only be accomplished if a good technical base is available at the Universities. It is therefore important to introduce computer science and software* technology curricula at the Universities. These curricula should be application oriented and contain a significant portion of practical work on computers. There is some danger, that computer science curricula are dominated by mathematicians. It is felt here, that a good combination of computer and software* courses, electrical engineering courses, mathematics and logic courses, economics and management courses and some work in an application field should form the core of computer science.

In order to keep the contacts with the international research community, research work in the area of computer science has to be conducted. The active participation at international meetings and the cooperation with research institutions in other countries are the prerequisites for the continued involvement in state of the art research projects. It is the obligation of the research organisation to critically reflect the state of the art in computer science and to provide valuable inputs to the political decisionmaker in relation to the state of the art of computer and software* technology.

Teaching computer and software* technology without the possibility of practical work on the machine is a dangerous undertaking. Since the lectures tend to become too theoretical, the student wil not grasp the elementary concepts and might shy away instead of developing a positive attitude towards this new technology.

Therefore any software* education initiative must be supported by an initiative to provide the necessary computer hardware* for the practical software* training on the machine.
Personal Computers and Professional Workstations form the recommended hardware* base for this practical training. Because of the good cost/performance ratio, their reliability and maintainability and their user friendlyness they are to be preferred over big central data* processing machines.

The low price of the modern personal computers makes it possible that these machines are used in even a very small business. Stock control, cost accounting and similar applications make it possible for the small businessman to take advantage of this new technology and improve his productivity over competitors not using this technology.

Computer and Software Training must therefore be included in any curriculum of Vocational Training Schools. Emphasis should be placed on the application of computers in the particular domain. The practical training on computers must play a dominant role in the computer education. Students should be taught to analyse their business activities in order to open their mind for possible new computer applications.

A special program* on software* development should be organised at the vocational school level. This is the topic of the next section.

Education of software* Engineers

The primary objective of this section is to present a detailed outline for a set of courses in core areas of software* engineering. These courses are designed for students who have finished their general education and are looking for a sound vocational training in the area of software*, as needed by government organisations and industry.

In developing this curriculum, a number of assumptions were made:

- scftware* systems are always components of larger hardware-software*-people systems.
- software* development requires more interaction and communication among people than in many technological endeavors
- the intellectual foundations of software* engineering are computer knowledge, application knowledge, technical management skills and communication skills.

The following core curriculum represents the minimal set of courses needed for practical work in the area of software* engineering. All theoretical classes are supplemented by practical work in a computer laboratory in order to bridge the gap between theory and practice. If a students wants study commercial data* processing applications or technical applications only, he can select the course "Commercial Data Processing" or "Real Time Systems" only. However, if time permits, it is advantageous to take both courses.

Overview:

	Class	Lab
Introduction to Computer Programming	40	80
Programming Methodology	40	80
Commercial Data Processing or	40	80
Real Time Systems	40	80
Project Management	40	-
Case study	-	200
Total	200	520
	(160)	(440)

The material presented in this curriculum on software* engineering is sufficient for a one year training program. As an alternative this software* engineering curriculum can be combined with a training program* in some application area (e.g. Accounting, Electrical Engineering, Industrial Engineering) of about the same size. Together this will be sufficient material for a two year training program. In this case we propose, that the two programs are interleaved and spread over the two year period.

Detailed Course Description:

Introduction to Computer Programming

Duration: about 40 hours of classwork and about 80 hours of laboratory work on a personal computer

Objective:
- to understand the basic methods of programming
- to understand the functional units and components of a computer
- Mastery of a programming language* (e.g. PASCAL*, BASIC* FORTRAN* etc.)

Course Contents. Programming: Algorithms and steps in algorithm*ic Problem solving, Flowcharting, Basic concepts in the programming language, variables, input-output, statements and expressions, conditional statements, loops, data* types
Services of an operating system, source and object representation of a program, compilation and assembly
Basic building blocks of a computer, Central Processing unit, storage units, input output units, Internal representation of data, binary arithmetic

Laboratory: Parallel to the class work the student has to develop programs in the chosen programming language such that the class work is well supported by the lab work.

Programming Methodology

Duration: about 40 hours class work and about 80 hours of laboratory work

Objective: capability for the architectural and detailed design of software* systems, including data* design* techniques. to evaluate the quality of a given design, elements of programming style, basic testing techniques

Contents: Design principles: Information hiding, data* and control locality, decomposition criteria, concurrency successive refinement, design* representations
Concurrency: Mutual exclusion, semaphor variables,

critical regions, event variables, messages, resource sharing.
design* evaluation and testing: Assessment of data structures, walkthroughs, Implementation tools, Test case design, test data* generators, regression testing, Documentation techniques, manual preparation

Laboratory: In the Laboratory the students should work in small groups (about 3 members) and implement a little software* project which has been specified by the teacher. Care is taken that the design* documentation and the user documentation is well prepared. A detailed test report has to be generated.

Commercial Data Processing

Duration: 80 hours classwork and about 80 hours laboratory work

Objective: Basic techniques of systems analysis in the commercial world, Data handling and data base design.

Contents: System Analysis Methods: HIPO, SADT#, Interviewing techniques, Functional Specifications Man Machine interfaces in the office, text processing and data* preparation on a personal computer Introduction to data* management, Functionality of Data Base Systems, Data Access Methods, File Design, Data security and recovery, Auditing of Database systems, Restart and Recovery

Laboratory: Team work, Design of a simple commercial application package, including systems analysis data* design* and implementation. Care should be taken that the documentation is up to standards

Real time Systems

Duration: 40 hours lectures, 80 hours lab work

Objective: Design and implementation of real time systems, such as process control system and systems for the control of discrete manufacturing and production control

Contents: Characteristics of process control systems, data* collection, interfacing instruments to a computer, basic concepts of control, real time programming, reliability and safety of control systems, man machine interfaces in the control room,
Discrete manufacturing, Computer Aided Design, Production Control Systems

Laboratory: Students, who select this course should have the possibility to implement a software* system for real time control on a small experimental pilot plant. Such a plant can be set up with a small number of cheap instruments and can be controlled with a personal computer.

Project Management

Duration: 80 hours of class work

Objective: To prepare the student for the task of Project Management and Economic Evaluation of Software Projects

Contents: Project Planning: Software Lifecycle, Resource and Schedule Estimation, Acceptance Criteria
Project Organization, Staffing, Project Monitoring, Human Factors,
Prototyping, Reporting, Technical Communication, Oral Communication, Report Writing, Presentation Techniques, Cost of Documentation
Software Economics: Cost effectiveness Analysis
Cost Estimation Techniques, Cost Factors,
Documentation: User Documentation, System Documentation, Maintenance Documentation, Standards
Legal Aspects: Legal Agreements, the Software Development Contract, Terms and Conditions, List of deliverables
Privacy of Information, Legal requirements

laboratory: There is no specific laboratory work included in this course. The material which is presented in this class should be applied in the practical project work

Case Study

Duration: About 200 hours of practical work

Objective: Specification, Implementation and Management of a realistic software* project

Contents: During the project work the material which has been covered in the classes should be applied in a realistic software* development environment. Work in teams, including project management and documentation of industrial standard

2.3 Standards

Many software* organisations are finding that the setting of standards for hardware* selection, communications, operating systems, programming languages, documentation, project management etc. has a significant payoff. The compatibility and the quality of the software* is increased, but more important for developing countries, the training and maintenance effort becomes much more effective.

On the other side, the stabilising effect of standards can also hinder progress. In a dynamic field like data* processing, new hardware* devices and software* procedures are continually developed. The benefits of taking advantage of these new developments must be carefully evaluated in relation to the costs, which result from the modification of existing standards.

The setting of standards is thus a delicate task, which has to go on continuously. In the light of new developments, it must be carefully assessed which standards are to be rigorously enforced and what areas are not yet ready for standardisation.

In this section we present some guidelines for the development of standards in the following areas:

- Programming Languages
- Operating Systems
- Computer Hardware
- Communication

Standardisation in the areas of software* project management, quality control and documentation will be discussed in subsequent chapters.

Programming Languages

In the short history of computing, many different programming languages have been developed. Although most of the languages did not gain wide acceptance, there are still so many different programming languages around that there is a definite need for standardisation.

The introduction of a new programming language* requires a significant effort, particularly in the area of education and maintenance. Programmers have to be trained in the new language and it takes some time until sufficient experience has been gained to make good use of the language. The software*, which is written in the new language* must be maintained, that is modified and enhanced. Therefore a generation of "maintenance programmers" must also be trained in this new language. Compilers for the new language* must be installed and some systems programmers must take care of the interfaces between the compilers and the operating system at hand. Furthermore, the introduction of the new programming language* increases the incompatibility problem in the area of application software*. It requires additional effort to combine application packages, which are written in different languages. Sometimes it is necessary to rewrite part or all of an application in order to integrate this application into an existing environment.

On the other side, there has been considerable progress in language* development over the past twentyfive years. The first high level languages were designed for engineering and scientific problem solving on large central machines. The commercial applications had different language* requirements for data* manipulation and thus gave rise to design* of some other programming languages. In the course of the years, more has been learned about language* design, such that a new generation of programming languages based on these new insights, has been developed. With the advent of the Personal Computer, the ease of use of programming language became an important factor, such that another set of languages, which concentrate on the ease of use in an interactive environment have become popular.

Besides these general trends, there is also some active marketing in the area of languages by the computer hardware* manufacturers. Since the effort to introduce a new language* is very significant, a user organisation, which is knowledgeable in the software* of a particular manufacturer is not likely to change to another manufacturer without good reason.

In the subsequent section we will discuss some of the more important programming languages and try to give some advise in relation to their usage.

BASIC*

The Programming Language BASIC* has been developed some fifteen years ago with the explicit goal of making it easy for the novice programmer to use the computer. Since it is not difficult to implement the BASIC* language* on a small computer, this language has gained very considerable support from the small computer industry. It is by now the most widely available language* on small personal computers.

The "simplicity of use" of BASIC* is not achieved without a high price. The BASIC* language* does not support the concepts of types and procedural and data* abstractions very well. It thus is difficult to write large and complex programs in BASIC*. Since the concepts of the first programming language* do have a decisive influence on the thinking pattern of a person, it is difficult to retrain a programmer who has only worked in BASIC* to take advantage of modern programming concepts.
BASIC* as a first language* is thus dangerous. However, considering the fact that most small computer manufacturers support BASIC*, it will be difficult to ignore BASIC* on the market place.

PASCAL*

The programming language* PASCAL* has been developed for teaching computer science. It is a small language* with a clean conceptual structure and a straightforward syntax. PASCAL* provides extensive support for data* typing, which is a fundamental concept in computer science. Having been designed for the educational market, it is missing some features which are important in the commercial market, e.g. the need for separate compilation and exception handling.

PASCAL* is also very successful in the Personal Computer Market. The general acceptance of PASCAL* as a teaching language is the motive for many computer manufacturers to support this language* in their product line. In the last few years, some dialects of PASCAL* which provide some additional functionality have been developed. However it is wise, to stay with the standard PASCAL* language* in order to stay compatible with the extensive PASCAL* software* market.

FORTRAN*

FORTRAN* is the "oldest" of the high level programming languages. Some twentyfive years ago it was designed for engineering and scientific problem solving. Since the language* can be implemented efficiently, it has gained significant support from the engineering community with the result, that a vast amount of scientific software* has been written in FORTRAN* and is available on the worldwide software* market. In order to eliminate some of the awkward features of the original FORTRAN* language, it has been modified and standardized. Even nowadays, the bulk of the engineering and scientific software* is still written in FORTRAN* (the dominant version is now the standardized FORTRAN* 77) and practically all major computer manufacturers support this language. In the world of engineering problem solving, FORTRAN* is the most important language* and it is speculated here that the dominance of FORTRAN* is this application area will prevail into the forseeable future.

COBOL*

COBOL* is also a language* of the first generation. It pioneered the development of data* description facilities and has become the most important language* in the commercial data* processing market. In contrast to FORTRAN*, which is concentrating on the formula manipulation, the expressive power of COBOL* is in the area of data* handling, file* manipulation and input output. COBOL* programs are to a certain extent self documenting,

since the language* contains many meaningful (and long) keywords. COBOL* has been standardized as early as 1960 .
COBOL* does hold the same position in the commercial data* processing market as FORTRAN* in the engineering market.

ADA++*

In the last fifteen years the field of embedded computer applications has grown considerably, but there has not been a single programming language* which has dominated this application area. As a result many different programming languages and dialects have been used for real time computer applications and process control. In large organisations, like the US Department of Defense, the use of many different language* resulted in a tremendous software* maintenance problem. In order to reduce this maintenance effort a decision has been made to develop a new programming language* for embedded computer applications which somehow combines the good features of the available languages. The development effort for this new language* started around the middle of the seventies. About three years ago the language* definition has been completed and the first compilers are appearing on the market now. Since there is a definite need for a language* for real time programming and there is a very powerful sponsor-the US Department of Defense-the success of this language* is probable. However, it will still take a number of years, until this language* is generally available.

C*

In the past, most operating systems, i.e the software* which controls the operation of the computer hardware, have been written in low level assembler languages. None of the mentioned programming languages provides the expressive power and efficiency, which is required in this application area. In conjunction with the development of UNIX+, an operating system which has been designed and implemented by Bell Labs, a new systems programming language, C* has been defined and used to implement the system software* of UNIX+. With the increasing popularity of UNIX+ in the personal computer market, C* is gaining considerable support as a systems programming language.

Conclusion on the topic Programming Languages

Considering the present state of programming language* development, it is recommended to promote the following standardized languages:

For the educational and training market: PASCAL*

For the commercial market: BASIC*, COBOL*

For the scientific market: FORTRAN*

For system programming, particularly on UNIX+: C*

There should be no restriction for the use of other languages, e.g. LISP*, PROLOG* etc. in the research environment.

The further development of the programming language* ADA++* should be carefully observed. As soon as there is a definite acceptance of ADA++* in the higher developed countries, ADA++* should also be introduced as a recommended programming language.

Operating Systems

The term "operating system" refers to the software* which controls the execution of programs. It provides services such as resource allocation, scheduling, protection, error management, input output control and data* management. Although operating systems are predominantly software*, it is possible tc implement parts or all of an operating system in the hardware.

At the time of the large central machine, the operating systems were supplied by the hardware* manufacturers and delivered with the hardware. One of the best known operating systems of that time is the IBM Operating system OS 360. It is a large monolithic operating system providing all of the services mentioned above. Similar Operating Systems have been provided by all major computer manufacturers. However the interfaces between these operating systems and the application software* are specific to a given manufacturer. This difference is a source of incompatibility as soon as services of the operating system are required, e.g. for input output, data management etc..

With the advent of the Personal computer the era of the commodity operating system started. Nearly all successful Operating systems for the Personal computer market have been developed by companies, which are independent from the hardware* manufacturers. The same operating systems runs on a number of different machines, thus providing the base for a degree of compatibility of the application software* which has not been achieved before. With minimal modification a given piece of application software* can run on a number of different machines from different manufactures, provided they all use the same operating system.

Since it is recommended in this report to concentrate on the small computer market, some standards for operating systems have to be established.

In the small computer market we can distinguish between two classes of operating systems, the single task operating system and the multitask operating system. A computer, which is equipped with a single task operating system can only perform one function at a+ time. On the other side, the multitasking operating systems provides the environment for the parallel execution of a number of programs. Given, that a system supports multitasking, there is only a small step to the multiuser support. Although quite a few single tasking and multitasking operating system have been developed by different manufacturers, only three of these operating systems have been evidently successful on the market place.

CP/M

This is a disc operating system for microcomputers produced by a company named Digital Research. CP/M stands for "Control Program Monitor". Versions of this operating system are available from a number of different sources for a variety of microcomputers. Nowadays, more than hundred different computer manufacturers offer CP/M with their equipment.

CP/M is a single user single tasking operating system,i.e. it supports only one user at a time doing a single program* execution. It provides the following services:

- file* management
- Input/Output support
- run time support for application programs
- error management

A variety of language* processors have been developed for CP/M, among others

- BASIC*
- PASCAL*
- COBOL*

The amount of application software* which runs under CP/M is very large, ranging from simple textprocessing software* to all kinds commercial and scientific packages.

CP/M has only one rival in the single user single task market of comparable popularity-- the MS/DOS Operating system from Microsoft. The functionality of MS/DOS is in line with that of CP/M.

UNIX+

UNIX+ is a multitasking, multiuser operating system, which has been developed by Bell Labs some ten years ago. With the introduction of powerful persoral computers and workstations this operating system is becoming a standard for the multitaking-multiuser market. In addition to the standard features of an operating system UNIX+ supports a hierarchical file* system. Significant amounts of application software* have been developed under UNIX+, particularly in the area of textprocessing, software* development tools, languages etc.. The UNIX+ operating system is described in some detail in the appendix.

Conclusion on operating systems

It is recommended here that the following standards for operating systems are considered

CP/M or MS-DOS as a single user, single tasking operating system
UNIX+ as an operating system for the multitasking, multiuser market.

Computer Hardware

Although this report is mainly concerned with guidelines for the software*, it is also necessary in this context to comment on hardware standards and developments. The explosive growth of the small computer market has -- within a period of five years -- already led to the development of two generations of machines with widely differing capabilities. The first generation of microcomputers was designed on the basis of the 8 bit microprocessor, i.e. information is processed in chunks of 8 bits. The new generation of machines processes information in 16 and 32 bit units. Since this makes the machines much more powerful it is recommended here to standardize on machines of the latter kind.

Communication

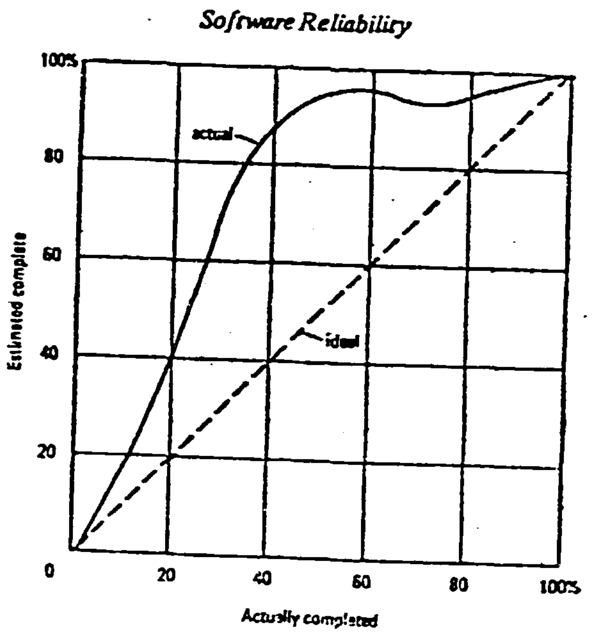
Although the field of communication is also outside the scope of this report, it is important to assess the future developments in the communication market and its relationship with the computer and software* industry. It is to be expected that the markets for computer and communications equipment are going to merge in the near future. It is therefore wise to closely cooperate with the planning authorities for the communication policy and to consider the formation of a joint committee for the establishment of standards which relate to both fields.

3. ORGANIZING A SOFTWARE PROJECT

The successful development of a software* product requires a sound management approach and technical expertise. This chapter is concerned with the management aspects of a software* project. The following chapter contains technical advise.

The usual management methods are planning, organisation and control. One reason for the frequent failures of software* management is the difficulty of adapting these techniques to software* projects. In the following section we will therefore characterise some of the difficulties which are typical for software* management.

If a comparison is made between the production of software* and a more conventional product, then the first great difference is the visibility of the result. The software* end-product consists solely of a set of carefully documented instructions for the computer -- there is no tangible software* product. The supervision effort required in determining development progress can be comparable with the development effort. A subjective estimate thus has to be made on the advice of the software* developer. The following figure shows a typical example, which may be often observed in practice, of how such an estimate corresponds to the actual situation.



/Kopetz, 79/ p.98

The development of conventional products is constrained by the laws of nature between relatively narrow limits (for example, the properties of materials), whereas the limits for software* are set by complexity and the ability of the human intellect to cope with it. The constraints due to complexity are very difficult to explain and quantify for people, who are not experienced in the field of software* development. It is therefore necessary that each computer specialist be highly self-critical and be aware of his own limitations in any situation. The lack of physical constraints is also responsible for the often incorrect view, that software* is easy to change, does not require a long development time and can easily be made to fulfill new conditions.

The rapid advances in both hardware* and software* make the software* planning task particularly difficult. By the time that an extensive software* project has been successfully concluded, economic grounds alone preclude a similar project on the same software* and hardware* basis. The result is that experience gained on an early project can only be adapted to a new project with difficulty.

The development of a software* system is a unique process as opposed to routine mass production. As with the construction of every unique product, it is difficult to establish the usual norms for progress and productivity. This may also be the reason for the often extremely poor documentation and maintainability of software*, since it is easy to underestimate the effort required for documentation by adopting the attitude that it is only for a single instance anyway.

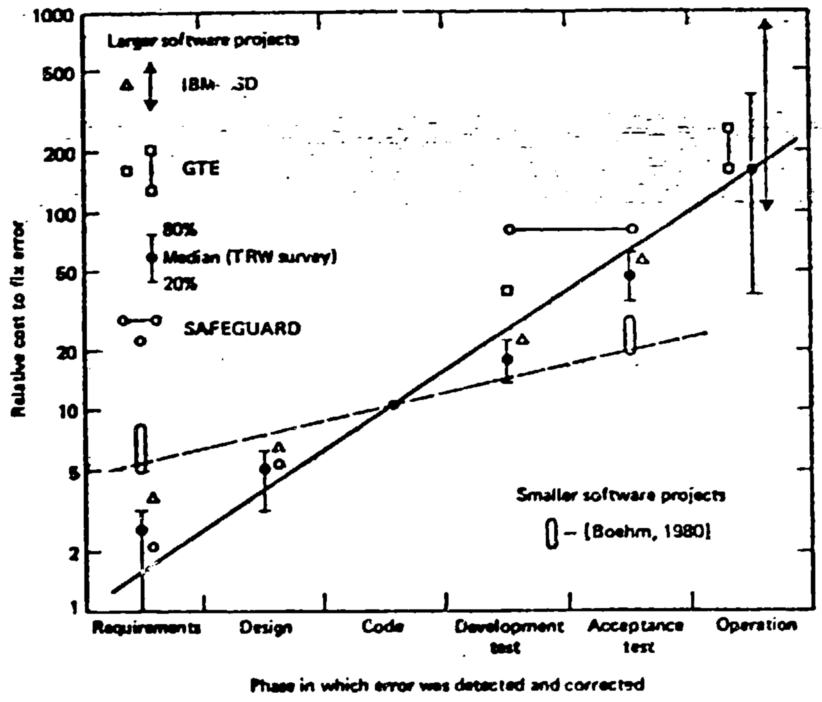
The success or failure of a project depends to a large extent on the personnel involved, due to the unusual difficulties of planning and control already described. The variation of ability between individuals is, however, particularly pronounced in the software* field, variations of 1:10 and more not being unusual. Software development requires creative personnel who can work with accuracy. However, creativity is often connected with personality traits which can led to problems in personal relationships. Any formal EDP training must be supported by project work that is at least as intensive in order to gain full benefits. Due to the rapid expansion in the field, however, it often happens that the successful project worker is assigned to management tasks and directly after gaining the relevant experience is lost to software* development. This danger is particularly acute in less developed countries.

This chapter is to give some advice on the organisation of software* projects. In the following chapter we will present a model for the subdivision of a software* project into a number of distinct phases. In the following section we will present some methods for effort estimation and the assignement of the overall effort to the phases introduced before.
The Team Organization will be the topic of the next section before putting everything together in an integrated planning system for project control.

3.1 Project Phases

In this chapter we introduce the basic phases of a software* project and discuss the scope of the activities in each phase. The model, which will be presented, is called the "Waterfall Model of Software Development". It partitions the Software Development Process into a number of distinct phases. Each phase is terminated by a verification and validation (VV) activity. Verification refers to the consistency between consecutive phases. Validation refers to the consistency between the phase and the real world problem statement.

This verification and validation activity is required in order to reduce the probability of an error being introduced during the work on the given phase. Experience has shown, that the cost for the elimination of a software* error increases substantially with the number of the past phases involved.



/Boehm, 1981, p.40/ Increase in cost-to-fix or change software throughout life-cycle

The Waterfall Model, as discussed by /Boehm 1981/ distinguishes between the following eight phases in the life cycle of a software* product:

(1) Feasibility
Determine the overall goal of the software* product and evaluate the potential product in relation to other alternatives, e.g. solutions without the use of a computer. This phase has to include an economic evaluation of the planned software* project, a rough cost estimation and a benefit analysis.

(2) Requirements
In this phase the requirements for the planned software* product are established. This includes functional requirements, interface requirements and performance requirements. It is of utmost importance, that the end-user participates in the establishment and validation of the requirements.

(3) Functional Specification
In this phase, the functional design* of the system architecture is undertaken. Considering the requirements, which have been established in the previous phase, the system functions are specified and a set of components (subsystems) and the interfaces between the components are defined. Care must be taken, that the proposed hardware* software* architecture will meet the performance requirements specified above. At this time a draft of the user manual has to be written.

(4) Component Design

In this phase, each component is decomposed into a set of programs, i.e. a sequence of about 100 executable statements in the given programming language. Care must be taken that the interfaces between the programs are defined and verified against each other and against the product design. The algorithm*s and data* structure for each program* has to specified during this phase.

(5) Coding

In this phase the actual coding of the programs, which have been specified in the previous phase, is performed. Each coded program* must be tested against the specification which have been developed in the previous phase.

(6) Integration

In this phase the tested programs are integrated in order to generate the components specified in phase number 2. The components are then integrated in order the generate the complete software* system.

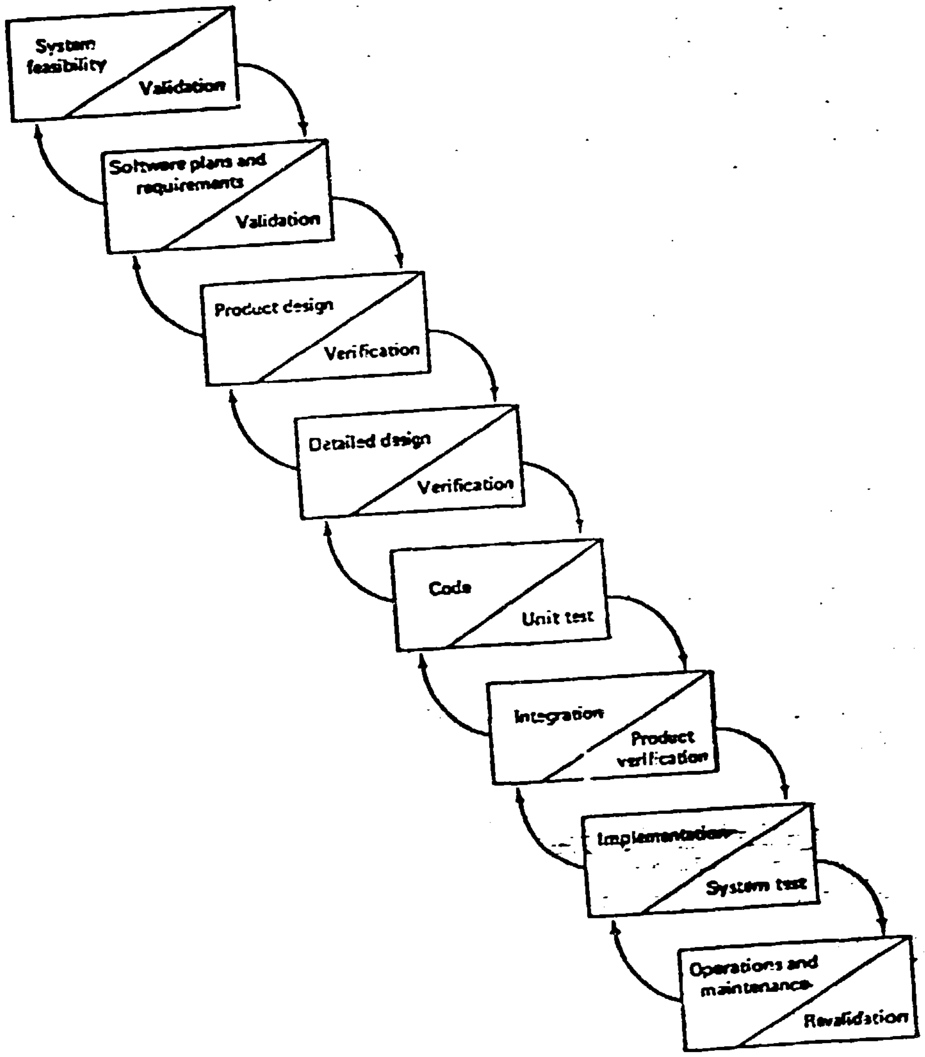
(7) Implementation

The software* system, which has been integrated and tested in the previous phase must now be implemented in the user-environment. The data* conversion, installation and training of the user personnel is part of this phase.

(8) Maintenance

Every successful software* system will have to be modified as the real world requirements change. During the life time of a software* product, these modifications will probably require more resources than the original software* development process.

A graphical representation of the Waterfall Model is given below



/Boehm 1981/, p.36

The disciplined software* development approach, as outlined by the Waterfall model, requires a good a priori understanding of the problem to be solved. Otherwise, a considerable amount of effort, which is spent during the early phases, can be lost if, at a later phase (e.g. the integration phase) the design* cannot be implemented as planned.

If there is no good a priori understanding of the problem, an incremental development strategy is the preferred alternative. In this strategy only the essential subfunctions of the system are developed in the first version of the system. After the viability of this reduced system has been established, the additional functions are added step by step. The development process for the essential subfunctions can also proceed according to the Waterfall model.

3.2 Software Effort Estimation

An estimate of the effort for a given task is a prerequisite for any planning activity. It will be clear by now, that software* effort estimation is an extremely difficult matter. However, it is necessary if a realistic project plan for a software* project is to be made. In many ways, effort estimation and control is the heart of software* management.

In our effort estimation we will measure the effort in the time needed (man-month) in order to get a project done. The cost estimation is a straightforward extension of this method, just multiplying the time by the current rate for a man-month and adding the additional expenses, e.g. computer time needed, clerical assistance, travel cost etc..

The big difficulty in software* effort estimation is the specification* of the size and complexity of a task in a metric which is generally accepted and usable for further analysis. Up to now, this metric is still the source code instruction, a line of code in the programming language* chosen. Although this metric is up to a lot of criticism, no better alternative for measuring the size of a software* task has been generally accepted. Software effort estimation can thus be broken down into the following activities:

(1) to derive the size and difficulty of a software* task from the functional specification

(2) to calculate the time required to perform the given task with the human and technical resources which are available

(3) to distribute the calculated time effort over the development phases outlined in the previous chapter

(4) to generate detailed plans in order to initiate, monitor and control the progress of the project.

Size Estimation

Estimating the size of a software* product relies heavily on the judgement of experienced performers. The software* analyst, or estimator, normally breaks the total job into elements that are estimated separately and then summarized into an estimate for the total job. The estimating analysis and synthesis may appear as a mental process or may involve an explicit algorithm*. In either case, an empirical database should be used as an objective reference. It is up to the estimator to use his judgement to account for the differences.

In general, we can distinguish between the following estimation methods:

(1) Top Down Estimating

The estimator relies on the total size or the size of large portions of previous projects that have been completed to estimate the size or of all or large portions of the project to be estimated. Historical data* coupled with experience and intuition is used to account for the differences between the projects. Among its many pitfalls is the substantial risk of overlooking special or difficult problems that may be buried in the internals of the project tasks.

(2) Bottom Up Estimating

The total job is broken down into relatively small work units, until it is reasonably clear how and with what kind of effort these units can be implemented. Each task is then estimated and the sizes are pyramided to get the total project size. An advantage of this technique is that the job of estimating can be distributed to the people who can do the work. A difficulty in this estimation method is the missing total view of the project. Parts, which are common to different units tend to get overlooked.

(3) Standards Estimating

The estimator relies on standards of size, which have been systematically developed. These standards then become stable reference points, from which new tasks can be calibrated. This method is accurate only, when similar work has been performed repeatedly and good records are available. The pitfall is that software* development is normally not performed repeatedly.

It is good practice to apply more than one estimation technique in order to cross check the estimate. The result of the estimation procedure* should be a table, which contains the main units of the software* system, their estimated sizes in source language* instructions and the difficulty in some form of complexity rating as discussed below.

Complexity Rating

The following software* categories for complexity rating have been selected based on experience /Wolverton 1972/. These software* categories refer to functionally different kinds of software* entities with different effort characteristics.

(A) Algorithmic units, which perform strictly algorithmic* (logical, numerical etc.) calculations without any consideration for execution time, input output or large data* management

(C) Control routines, which control the flow of execution and are non time critical.

(D) Data management routines, which manage data* transfer within a computer and its peripheral devices

(I) Input Output routines, which transfer data* between a computer and its environment.

(P) Pre or Post Algorithmic Processing, which prepares and manipulates the data* for or after algorithmic* processing.

(T) Time critical processing, which is highly optimized machine dependent code.

In each one of these six categories we can distinguish between the following difficulties:

Easy Medium Hard

The following table can serve as a rough reference for the relative effort required for each one of these categories. This table has to be modified as experience accumulates.

Degree of	Software Category					
Difficulty	A	C*	D	I	P	T
Easy	1.0	1.4	1.6	1.2	1.3	5.0
Medium	1.3	1.8	2.1	1.6	1.5	5.0
Hard	1.5	2.0	2.3	1.8	1.7	5.0

If we multiply the estimated size of each software* unit with the corresponding degree of difficulty, we get the normalized size of the units.

Environmental Factors

The effort, which is required to produce a given piece of software* depends on the product per se (normalized size) and on environmental factors of the software* producing organization. Some examples of environmental factors, which do have an influence on the time required to complete a given task are:

- Qualification of the development Personnel
- Experience of the development Personnel
- Development System at hand
- Concurrent Hardware/Software Development

Although all of these factors are important, the qualification and experience of the programmers seem to have the most significant influence. The following table gives some indications of the differences which have been observed a number of times

Experience of	Qualification of Development Personnel		
of Dev.Pers.	below average	average	above average
little experience	.5	.7	1.0
average	.7	1.0	2.0
very experienced	1.0	2.0	3.0

If a powerful software* development system is available the productivity of a programmer can be increased by up to 50 %. The concurrent development of software* and hardware* is normally a significant handicap, which can cut the productivity of software* development to half.

If we multiply the normalized program* size with the environmental factors discussed above, we get the Work Size of the Software

Job. In order to arrive at the time needed to implement this Job, we have to divide the Work size by the applicable productivity rate.

Productivity rate

A lot of experimental data* has been collected on software* productivity. However, considering the many factors involved it is very difficult to compare the productivity data* which has been accumulated on different projects with different people in different development environments.

Before establishing a local productivity data* base, which takes all the local factors under consideration, the following estimate of programmer productivity can serve as a rough first guideline:

Considering the Work size as the base, it can be expected that about 300 - 400 lines of source code per month can be produced by an average programmer. This time includes all activities in the following phases:

- Requirements
- Product Design
- Components Design
- Coding
- Implementation

The final documentation of the software* product is also included in this productivity number.

Phase Distribution

This section deals with the assignement of the development time to the different project phases introduced in the previous chapter.

Boehm /Boehm,1981/ gives following phase distribution for average software* projects:

Effort Distribution

Phase	Product Size		
	small (2KDSI)	medium (32KDSI)	large (128KDSI)
requirements	6 %	6 %	6 %
product design	16 %	16 %	16 %
component design	26 %	24 %	23 %
Coding	42 %	38 %	36 %
Integration	16 %	22 %	25 %
	100 %	100 %	100 %

1 KDSI 1000 delivered source code instructions, i.e all instructions which are written by the programmer, excluding comments.

The schedule will normally differ from the effort distribution. In the beginning of a software* project, during the requirements analysis and system design* phase, only a small number of highly experienced software* specialists will perform all the work. The component design* and coding can be distributed to a large number of professionals. Thus the first phases of a software* project will take longer than the corresponding ratio of the effort estimation.

The total schedule of a large software* project can be calculated according to the following formula:

Total schedule (in months) = 2 * SQRT(Work size (in thousand SI))

For small and medium projects (those less than 100 manmonths), this formula is not applicable.

The work distribution during the project duration will differ considerably according to the project phases. During the system design* a small group of experts should be in control of the complete design* task. Later on, during component implementation, the work can be distributed on a number of people.

3.3 Team Organisation

It has been shown in the previous chapter that the productivity ranges of the individual programmers are very significant factors in the effort estimation procedure. Personnel attributes and human relations activities thus provide by far the largest source of opportunity for improving software* productivity.

Normally, the project team is the preferred organisational structure for software* development. All the talents, which are necessary for the development of a software* system, should be present in this team. For the time of the project, the members of the project team report to the project manager and are freed from all other duties. The project team should consist of members of the software* development department and the user organisation.

Boehm /Boehm 1981/ introduces five basic principles for software* team staffing:

- The Principle of Top Talent
- The Principle of Job Matching
- The Principle of Career Progression
- The Principle of Team Balance
- The Principle of Phaseout

The Principle of Top Talent

The bulk of the productivity on a software* project comes from a relatively small number of highly qualified participants. If there is an alternative, it is superior to use fewer, but highly qualified people in order to get a software* project done. A number of studies have shown that the well known 20%/50% rule applies to software* development: 20% of the highly qualified people provide 50 % of the work.

The Principle of Job Matching

Although software* work is not repetitive, there is considerable opportunity to transfer the experiences gained on one project to another project of similar characteristics. This can improve the programmers productivity considerably. It is therefore important to carefully match the programmers profile* to the job profile.

The Principle of Career Progression

Since the software* field is growing rapidly, it is common practice to advance the good programmer into management. This can be a big mistake, since it is not definite, that a good programmer will be a good manager. On the other hand, some technical expertise, which has been available to the organisation is lost. It is important to provide career paths for technical experts so that they can achieve a high social standing without turning into management. A successful software* organisation relies more on technical experts than many other engineering organisations.

The principle of Team Balance

Software work is team work. System people and people from the user organisation must cooperate harmoniously in order to get the work done. It is a management duty to assign the personnel in such a way to the project teams that a balanced set of talents is available and no extreme personality traits dominate the team.

The Principle of Phaseout

If some extreme personalities dominate the team in an unproductive manner, it is important to phase these persons out of the team as soon as possible. Otherwise a considerable amount of the productive capacity of the team will be used in order to resolve these internal conflicts.

3.4 Project Control

A prerequisite for the effective control of a software* project is the availability of detailed project plans. It is assumed that the project manager is responsible for planning from the beginning of the project (requirement* analysis) until the delivery of the end product.

In the previous sections we have already discussed some of the techniques for structuring a software* project, for software effort estimation and workload distribution. We will now put these things together in order to generate a comprehensive project plan.

We distinguish the following sections in a project plan:

(1) Project overview
This section gives an overview of the project. It describes in short words the main objectives of the project, the user and development organisation and explains the structure of the plan.

(2) Phase Plan
The Waterfall model introduced in the beginning of this chapter can form the core of the phase plan. In addition to the project structure the phase plan must contain the effort, both man and machine, which is needed for the completion of each phase and the definition of some tangible products, which are produced at the end of each phase. Since these tangible products will normally

consist of project documents, the phase plan and the documentation plan will be closely related.

(3) Documentation plan

The Documentation plan defines and is used for the control of the Project Documentation. It is one of the most important plans of a software* project. The minimal set of documents, which have to be produced during a software* project are the following:

- Feasibility Study: The documentation of the economic analysis of the proposed computer application including a cost benefit calculation

- Requirements Analysis: The documentation of the requirements of the new system

- Functional Specification: The documentation of all system functions, including input and output procedures, logical data* base design, and the definition of the acceptance test

- User Manual: This documentation includes all the information which is necessary for the operation of the system. A first version should be prepared together with the Functional Specification.

- Program Documentation: It includes the information which is necessary for the modification of the the delivered software*.

(4) Test plan. This plan contains all testing activities, such as module tests, integration tests and acceptance tests.

(5) Organisation plan: This plan defines the specific responsibilities of each person participating in the project. It includes the estimated work effort and the start and completion date for each project task. The milestones in the organisation plan must be coordinated with the documentation plan and test plan, such that tangible results can be monitored.

(6) Installation plan
This plan includes all activities which are concerned with the installation of the proposed software* product. The important topic of training of the users personal can be either included in the installation plan or can be dealt with in a separate training plan. The installation plan must also contain all dates concerning the physical system installation.

(7) Reporting plan
This plan describes the reporting structure about the project,i.e. the reports to the project manager and the project steering committee. It is good practice to introduce two types of project reports, periodic reports and phase completion reports. The periodic reports contain all the activities which have been completed in the last reporting period as well as an outlook on the next reporting period. The question about potential problems affecting the progress of the project should be part of every project report. A good frequency for the periodic

report is about once a week. The phase reports are produced at the end of each project phase. They contain a comparison of the planned versus actual effort required for the phase in question.

These detailed project plans form the backbone of the project control. During the project, the project manager must monitor the progress of the project in relation to these plans. If a significant deviation between the planned and the actual progress of the project is observed, it is good practice to question all project plans and to iterate through the planning phase once again.

3.5 A case study for organising a software* project

In this section we want to give a practical example for the application of the effort estimation and project control techniques.

Let us assume, that a company wants to develop a new software* package for order processing. A feasibility analysis has shown that such a package could result in savings of about 20.000 US $ per year.

The package has to support the following functions
- order entry on an online terminal
- order processing
- data communication (transmission of the order data to the accounting department).
- report preparation

Based on the experience with systems of similar functionality and complexity, the following estimates for the program size are made:

(1) Order Entry
This subsystem must support ten different CRT formats and about 40 different input records. Some plausibility checks on the input have to programmed, as well as a number of accesses to the order product file, order file* and customer file:

Size estimation:

unit	size	complexity	number	total	norm
CRT output	50	IM	10	500	800
input proc.	20	PM	40	800	1200
file access	10	DM	10	100	210

The abbreviations in the colums complexity are taken from chapter 3.1 . Total stands for the total estimated size and norm refers to the normalized size, i.e. the estimated size multiplied by the difficulty factor from chapter 3.1.

(2) order processing
In this subsystem the order has to be analysed and checked for validity. The required papers for the warehouse have to be printed and an order confirmation has to be sent to the custormer.

Size estimation

unit	size	complexity	number	total	norm
order anal.	200	AM	1	100	260
warehouse pap.	400	I,P M	1	400	610
order conf.	250	I,P M	1	250	390

(3) Data communication
In this subsystem the data communication protocol* between the order entry machine and the accounting machine has to be developed.

unit	size	complexity	number	total	norm
protocol setup	200	I,T M	1	200	660
comm.error man.	300	I.T M	1	300	990

(4) Report preparation

In this subsystem about 15 different management reports have to be prepared.

unit	size	complexity	number	total	norm
report prep.	40	AE	15	600	600
file access	10	DM	10	100	160
report output	50	IM	1	50	80

If we compare the estimated sizes and normalised sizes of the four subsystems, we get the following results

subsystem	estim. size	norm. size
(1) Order Entry	1400	2210
(2) Order Processing	750	1260
(3) Data Communication	500	1650
(4) Reporting	750	840
Total	3400	5960

We now assume, that we have two programmers available, one beginner of average qualification and one experienced programmer with average qualifications.

programmer	norm size	human factor	work size
experience	3960	2	1980
beginner	2000	.7	2857

Let us assume that the productivity rate is about 350 lines of code per month. This gives a total effort for this project of about 18 Manmonth.

If we now look at the effort distribution, we get

product design	16 %	3 MM
component design	22 %	4 MM
Coding	40 %	7 MM
Integration	22 %	4 MM
Total	100 %	18 MM

We now can fix some of the milestone dates of this project:

Only the experienced programmer will work during the product design, such that after 3 month the product design* (functional specification) will be complete. This will be the first milestone.
The rest of the work will be done by the two programmers in parallel, such that the whole project will be completed after about 10 to 11 months.

Phase plan

This is a rather small software* project. We will therefore distinguish between the following phases

Feasibility to be done by the user organization

Requirements to be done by the user organization in cooperation with the software* development organization. According to chapter 3.2 the requirements phase will take about 6 % of the project work, i.e. about 1 Manmonth in this case. This effort is not included in the effort estimation procedure.

Product Design to be done by the software* development organisation
According to our estimate 3 Manmonths

Functional specification* 4 man month effort, to be completed within
two month by the software* development organisation

Coding 7 Manmonth effort, to be completed within 3.5 months
by the software* development organisation

Integration and Implementation 4 Manmonths, to be completed
within 2 months by the software* development organisation

Documentation plan

The Requirements Analysis Document must be completed after
1 month at the end of the Requirement phase.

The functional specification* document and a preliminary
version of the user manual must be completed after the
Product design* phase, i.e. 4 month after project start.

A preliminary version of the program* documentation has to
be completed at the end of the coding phase, i.e 9.5 months
after project start.

Test plan
The detailed procedures for the acceptance test will be
specified in the document "Functional Specification".
The component tests will be performed during and at the
end of the coding phase.
The acceptance test will be performed at the end of the
integration phase.

Organisation plan

The organisation plan states that one specific programmer will be assigned to this project for the first four months, and after that date the selected second programmer will join. The first programmer will act as a project manager. The key dates and milestones of the project are those of the phase plan.

Installation plan

After the functional specification* (four months after project start), the training of the user personnel will commence. (Note, that the preliminary version of the user manual is completed by that time). The implementation phase of this project will start ten months after project start with active participation of the user. In case new equipment has to be installed at the users site, this installation must be completed nine month after project start.

Reporting plan

Reports about the progress of the project have to be prepared every other week. At the end of each phase, a summary report, giving a management overview over this phase, will be provided.

The following time table gives an overview of the project

```
Time (in weeks    0         1         2         3         4
after start)      1234567890123456789012345678901234567890123456

Requirements      rrrr

Functional Sp.        ffffffffffff

component design                  dddddddd
                                  dddddddd
Coding                                    cccccccccc
                                          cccccccccc
Integration                                          iiiiiiii
                                                     iiiiiiii
```

4. The Software Development Process

The final chapter of these guidelines is concerned with the technical aspects of software* development. It is to be understood, that this chapter can only give an overview of the important technical topics. It is recommended that the reader refers to the abundant software* literature, part of which is referenced in the bibliography, for further study.

The main emphasis of this chapter is on the Quality Control aspects of software*. We will therefore develop some checklists for each phase in order to help the software* engineer in auditing his work.

4.1 Requirements Analysis

As already mentioned before, the first activity in a software* project is a feasibility analysis. Since such a feasibility analysis is not typical for software* -- it must be performed for any kind of investment -- it will not be discussed further in this report.
The result of the feasibility analysis is a cost benefit analysis of the proposed project and a coarse description of the objectives of the new project.

The requirements analysis takes this specification* of the project objectives as the starting point. The requirements analysis phase can be structured into the following segments

(1) review of the project objectives
(2) review of existing methods and procedures
(3) Preliminary specification* of the system requirements
(4) Analysis of the preliminary requirements
(5) Final specification* of the requirements

The result of the Requirements Analysis is the report on the System Requirements. The following checklist is provided in order to make certain that this report is complete:

- Objectives of the computerized system
- Analysis of existing methods, including a description of the environment, in which the system will ultimately operate. Includes rules and regulations, policies critical aspects of this application.
- Scope and penetration of the system as defined by organizational boundaries, plans for organizational changes, concurrent projects which might have an effect on this work.
- Description of the typical system user, its background experience, expectations and training requirements.
- A general information flow chart of the application, showing key inputs, outputs, volume estimates (average, peak, growth) and time constraints which are critical. Areas or origination and use of inputs and outputs is shown. The information flowchart displays major decision points in the application.
- Interfacing requirements. A detailed description of all system interfaces which are given by the environment of the new system. Possible changes in these interfaces mu't be investigated.

- Security and Safety requirements: Privacy and restricted access to sensitive data, reliability of the system and the data
- Go-nogo criteria. Critical design* constraints which must be met in order to install the system. This does not include performance parameters only, but also cost and development time.
- Statement of assumptions. Analysis in respect to the criticality of these assumptions.

Once the systems requirements have been established, these requirements must be validated. There are four criteria for evaluating the requirements:

(1) consistency: is there a conflict between some of the requirements ?

(2) completeness: Are there any functions which have not been considered? Are there any constraints which may have been overlooked?

(3) validity: Are the requirements needed to fulfill the objectives of the "wider" system?

(4) realism: Are the requirements realistic considering the given environment?

There have been a number of tools and techniques developed for the support of this phase. One of the best known methodologies for the requirements analysis phase and design* phases is SADT# --Structured Analysis and Design Technique. This technique will be described in some detail in the appendix.
An excellent overview concerning the different Methodologies for Requirements Analysis and Design can be found in /Wassermann et al,1983/.

4.2 Functional Specification

The Functional Specification is concerned with the process of going from the statement of the requirements to a description of the functions to be performed by the system. Since the functional specification* involves the external design* of the software, it is sometimes referred to as "product design".

The following checklist should help the designer in preparing a complete and consistent functional specification. A functional specification* must contain:

- Identification of the objects which are visible at the interface to the environment, including the attributes and relationships of these objects.
- Identification of the functions which will operate on these objects, including their domain.
- Detailed specification of the inputs and outputs of these functions including the formats and dialogue. The inputs and outputs must be referenced in the information flow chart of the requirements analysis.
- Detailed specification of the information which will be stored in the system. The inputs used to create, update or change this internal information must be identified together with the data* elements they contain.
- Detailed specification of the processing steps. Decision tables, algorithm*ic formula or some kind of Program Design language* can be used to represent these algorithm*ic steps.

- Starting from the volume and timing information of the requirements analysis, this section is concerned with the performance characteristics of the planned software.
- Specification of the procedures for system start up, restart and error handling. Data base recovery procedures after a system failure (e.g. by power failure)
- A data* dictionary defining all data elements, their meaning and representation at the external interface of the system. The plausibility checks for these data elements should also be included.
- The detailed procedure* for the acceptance test of the software. The user may be called upon to provide the test data* before a specified date.
- A chapter on the conversion of the "old" system to the "new" system.
- A project plan, as outlined in chapter 3 of these guidelines. A revision of the cost benefit analysis, based on this project plan must also be included.

4.3 System Design

In this phase the internal structure of the software is devised to provide the functions specified in the previous stage. The most important activities during the system design* are the partitioning of the system into subsystems and the design of the data* structures of the system.

Design is a creative activity. It is the link between analysis, programming and operations. On the one side, it must fulfill the functional specifications, whereas on the other hand it has to meet the given technical requirements. According to /Collins et.al 1982/ design* consists of the following steps:

(1) Understand the requirements
(2) Break the system into its basic components of data and processes
(3) Design the logical structure of the system
(4) Adapt this structure to the physical implementation constraints
(5) Specify the subsystems

In a data* oriented design, the first step after the analysis of the requirements concerns the definition of the logical data* structures. Starting from the inputs, outputs and internal data* elements of a system the relationship between these data* elements are established on the basis of the conceptual model of the application environment. This work is the basis for the design* of the logical data* structure. The operations on the logical data* structure are the starting point for the process decomposition. Processes are identified by following the flow of data* through the system. Additional requirements, eg. auditing requirements, will be implemented through additional processes.

The final result of the system design* is the system design report. It has to contain the following items:

- A general overview of the entire system
- Subsystem description: a short description of each subsystem including the input and output data* of each subsystem, the internal data* of each subsystem, the processing functions of each subsystem,
- A system structure diagram showing all subsystems and their interconnections.
- The description of all system data* elements and a cross reference of data-processes.
- The description of all processes and their inputs/outputs timing and volume analysis
- Methods for implementing the data* security *and privacy requirements
- A program* portfolio -- for each program* in the system, the System Design Report must contain all the information needed by the programmer, including
  * program* overview: a brief description how the program* fits into the overall system
  * A specification of the program* logic (algorithms*) and the external datastructures, the semantics of the internal states of the program
  * Layout of the physical format of all input output, including form design* and record definitions
  * Error handling information, including error detection requirements, error reporting strategy and restart information

* Testing strategy, both for the program* test and the integration test

System design* is an iterative process. Therefore it is necessary to evaluate a design* before it is frozen. This can be done by formal inspections or less formal design* reviews. Any design* should be evaluated in relation to the following criteria

(1) consistency and completeness in relation to the functional specification

(2) complexity of the the components and interfaces and component size

(3) stability of the interfaces in relation to possible change requirements

(4) observability and testability of the design

(5) performance characteristics in relation to the available hardware* environment

4.4 Programming

The programming effort starts with the analysis of the System Design Report and the Program Portfolio by the programmer. It normally consists of the following activities:

- Logical Analysis
- Design of the Control Structure and Internal Data Structure of the Program
- Desk Checking and Logic Review of the Program Design
- Coding and Documentation of the Code
- Compilation and Debugging of the Code
- Preparation of Test Data and Module Testing
- Updating of the User and Operator Instructions

Next to a well-conceived design, the programming style has a decided effect on the comprehensibility of a software system. There are, of course, many ways in which a program may be written so that the required data transformations are performed. It is the individual programming style that finally decides which alternatives are chosen and whether a program looks clear or confused. Programming style determines the readability and thereby also the complexity. Complexity is the main cause for software errors and the unreliability of the end product. The most important rule of programming style requires that a program be simple, clear and manageable. The clarity of the program can be improved, if some of the following rules are obeyed:

-Physical Layout: The visual organisation of the program text using separating lines and indentation has a very positive effect of program readability.

-Naming: The choice of variable names is very important for readability. While variable names should be chosen from a mnemonic and systematic point of view, they should also make reference to the meaning of the variable. Starting from a problem oriented base, the naming effort should be strictly controlled by the project manager.

- Processing: In the processing section of a program one should make use of tried and proven program segments, as far as possible. The control elements in a program should be taken from the limited set of constructs defined in the "Structured programming rules", i.e. sequence, <u>if</u> <u>then</u> <u>else</u> and <u>do</u> <u>while</u> . In many cases complicated branching can be avoided by the use of' simple logical operations on logical variables.
- Input Output: Many program errors can be traced back to badly laid out input/output interfaces or a poor check of the input data. Input formats should be uniform and as free as possible. Similarly, results should be presented with a meaningful text. The range of input values should be kept as small as possible and reasonable and should be checked in the program.
- Testing: Every program should be designed in such a way that it can be tested independently of the rest of the system. It is therefore important to provide provisions for the observability and control of the inputs, outputs and internal state variables of a program.
- Error handling: Every program should contain an error detection and an error handling section. Errors, which cannot be handled in the program should be reported to the calling program. The internal state of the program should be set in a consistent state.
- Commenting: It is not easy to comment a program meaningfully. A program may not only have too few comments, it may also have too many. Each program should start with a block of comments which should contain the following information:
  * program indentification, version and author
  * date of production and last modification

* function of the program
* accuracy statement (especially in real arithmetic)
* input output description by example
* example of calling sequence and parameter description
* error handling
* assumptions regarding environment and module limitations

Furthermore, branch points within a program should be described with reference to the algorithmic solution. Any program section that probably will have to be changed as a result of program modification should be highlighted.

4.5 Testing and integration

At the beginning of every software development some concepts of what the system should do are formulated. The validation of the software package constitutes confirmation that these concepts are in fact fulfilled. The validation is a continuing process through each stage of the software life cycle.

Even with simple programs only a vanishing small part of all theoretically possible input cases can be exercised during the test phase. This is why software testing can only show the presence of errors, never their absence.

In order to make the test phase effective, it is important to carefully select the test cases. Three approaches for test case selection may be taken: to check the specified functions of the program, to select the test cases on the basis of the input distribution of the expected application, or to determine the test cases on the basis of the internal structure of the actual program. In the following sections these three test methods, the functional test, the acceptance test and the structural test, will be treated.

During functional testing the specified functions should be tested individually and in combination. The software system is treated as a 'black box' by means of which expected results are calculated for given input values. The selection of relevant test cases on the basis of functional testing represents the most effective method of testing a program.

The acceptance test is based on the point of view of the actual employment of the system by the user. The choice of test cases is made by the division of the input space into application related input regions. Important factors in choosing the test cases are the problem specific input cases in the various regions of the input space. The chief advantage of the acceptance test lies in the complete coverage of the development chain through a close cooperation with the user.

In the Structural test, the test cases are selected on the basis of the internal structure of the program. This selection can be made according to the following criteria

- Every instruction in the program* must be executed at least once
- Every branch point should be tested in each direction, at least once
- All control paths have to be tested.

Even the test of all control paths is not sufficient for the complete coverage of the program.

Since none of the test methods is perfect -- as a matter of fact, there is no perfect test method known -- it is recommended to use a combination of methods for the test case selection.

The testing activities should be formulated in a test plan. The components of a good test plan are:

-A listing of all functions that are to be tested
-a description of the test strategy that will be used
-test criteria that must be fulfilled, such as percentage of branches, range of variables etc..
-error rate that must be achieved before the system can be classified as deliverable
-performance requirements expected for the complete system and its components, such as response time etc..

-a list of errors that the system must be able to tolerate
-guidelines for the documentation of the test results
-the expected results of the test cases
-procedures for sy :tem integration
-schedule for the test phase

Whereas with testing an attempt is made to establish confidence in reliability of a program* by means of a point-by point check of the input, another verification technique, the analytical program proving, concerns itself with program* per se as opposed to just the computational results using the program. The aim is to show by formal means that the program* in itself is correct The current situation with analytical software verification appears to indicate, that in the near future it will not represent a viable alternative to program* testing.

4.6 Maintenance

The analysis of the life-cycle of successful software systems has shown that the cost of keeping the software operational significantly exceeds the cost of the initial software development. The software effort which is needed to maintain an operational system is commonly referred to as the "software maintenance effort". The maintenance phase thus starts with the completion of the acceptance test and continues until the system is discarded.

Software maintenance is significantly different from hardware maintenance. In hardware* maintenance a hardware* unit which has changed its physical performance due to ageing or environment stresses has to be replaced by a new unit of the same type, i.e. the same initial performance, whereas software maintenance always implies the partial redesign* of an existing software product. This redesign* becomes necessary for the following reasons:
(1) repair of residual design* errors. Since it is not practically possible to exhaustively test or analytically verify all properties of a program, there is always a certain probability that residual design* errors be detected long after the system has become operational. However, it is to be hoped that an improved programming methodology will reduce this number of residual design errors.

(2) Adaptations and functional enhancements. Everv successful system becomes part of the environment it is destined to serve. Since the real-world environment is continually changing, the requirements of the system are also forced to change. "In real

life there is not a single, static well-defined problem but a constantly changing problem whose definition is being altered by information that actors recover from memory and by other information from the environment's response to actions taken". /Simon 1971/.
In order to keep a system operational, it must be changed so that it corresponds with these modified requirements. Examples of such changes in the requirements are:

-Modification of the real world function which is modelled within a system (e.g. implementation of a new income tax regulation in a payroll package).
-Availability of new hardware* (e.g. modification in the database software to use a new disk drive with improved cost performance).
-Performance enhancements (eg. improvements in the response time of a new on-line system).

Most of these changes are outside the control of the system implementor. There is no hope that an improved software development methodology can significantly reduce these change requirements. On the contrary, the increasing integration of computer systems within organisations will lead to a need for an increased responsiveness to change requirements. In the future, some organisations might only be capable of surviving if their computer systems can be modified quickly in response to a changing market place.

Appendix 1: The UNIX+ System

UNIX+ is a general purpose interactive operating system that has been developed by Bell Labs. The first Edition of UNIX+ was documented in a manual authored by Thomson and Ritchie from Bell Labs in November 1971. It was implemented on a Digital Equipment PDP 11/20 computer. Since that time UNIX+ has been implemented on a number of different computers ranging from micros to big mainframes. It is now regarded a standard operating systems for the more powerful micro-computers.

In the very short history of the mass produced personal computer experience has shown that only a small number of standard operating system penetrate the market. Since, at the moment, there are no real alternatives to UNIX+ as a standard operating system for powerful personal computers, it must be expected that UNIX+ will become much more popular with the introduction of more powerful mass produced personal computers. Already nowadays, UNIX+ is supported by many different suppliers.

As far as possible, UNIX+ tries to hide the characteristics of a specific machine. This results in a high compatibility of the software written for UNIX+.

The main features of UNIX+ are:

- A hierarchical file* system which is independent of the parameters of the physical storage device.
- Compatibility between the input/output and the file* system

- The ability to create asynchronous processes
- A flexible command language* which can be tailored to the requirements of a given application environment
- A large amount of software from independent software suppliers, including all major programming languages, relational data* base systems, text processing software and a variety of application packages.
- Communication support to connect UNIX+ systems together

In the following we will discuss the two most important features of UNIX+, the file* system (which includes the input output system) and the command language.

The UNIX+ file* system

A file* system provides a facility to store information by name. UNIX+ distinguishes between three kinds of files

Ordinary files

The ordinary file* is the standard for the storage of information from a user program. No particular structure of the information is expected or supported in ordinary files. An ordinary file consists simply of a sequence of characters, with lines demarcated by the newline character. Binary program* files are sequences of words as they will appear in the core memory of the computer. If an application program* expects any structure of the file, it is in the responsibility of the application program* to generate this structure and interpret this structure. From the point of the UNIX+ file* system, files do not contain any structure. Even the structure imposed by the physical storage medium is hidden from the user.

Directories

A directory is a file* that holds the names of other files (or other directories). It thus provides the required mapping between a filename and the file* itself. Each user has a directory of his own files. He may also create subdirectories to contain groups of files which are related to each other. A directory behaves exactly like an ordinary file, except that the operating system controls the write access to this file. Provided the user is privileged to do so, he can read a directory file but may not modify it.

The starting point of the file* system is the root directory. All files in the system can be found by tracing a path from the root directory to the next level of directories and so on until the required file* has been located. The file system of UNIX+ supports thus a hierarchical structure of files. Files are named by sequences of 14 or fewer characters. When the name of a file* is specified to the system, it may be in the form of a path name, i.e. a sequence of directory names. A file* in UNIX+ does not exist in a particular directory. The directory entry of a file* consists merely of its name, its attributes, and a pointer to the information of the file. Thus it is possible to link the same file* to different directories. The operating system maintains several directories for its own use. Some of these directories contain the programs which are for general use. The execution of these programs can be invoked by specifying the file* name of the appropriate program* file. These file* names correspond to the system commands. By writing new programs and linking them to the appropriate directories, a user can develop his own command language* tailored to his particular application environment.

Special files

Special files correspond to input output devices. These files can be accessed in the same basic way as ordinary files, though with some restrictions e.g. it is impossible to write to an input device. An access of a special files causes the activation of the corresponding device. Each disk drive, each terminal or communications line can be accessed through its special file. The access to special files can be protected, just as the access to ordinary files. The advantage of treating I/O devices and files the same way lies in the compatibility between I/O and file* access. No special programming is needed to redirect I/O to a file. The system calls to do I/O are designed to eliminate the differences between the various I/O devices and styles of access. There is no logical record size imposed on a file. UNIX+ does not provide for any file* locking.

The UNIX+ command language

UNIX+ provides a command line interpreter, which is called the "shell". It reads lines typed by the user and interprets them as requests to execute other programs. The simplest form of a command consists of the command-name followed by its parameters:

```
command-name par1 par2 par3
```

The shell separates the command name and the parameters into two strings. The first string is the filename of the program which is to be executed. This program* file* is sought, brought into main memory and started for execution. The arguments which have been collected by the shell are made available to the program. When the program* is finished, the shell continues its own execution and informs the user with a prompt character. If a file* with the name of the command cannot be found, then the shell reports an error to the user.

When a user starts working on his terminal, the shell declares the user terminal as the standard input output device. However, the shell can dynamically change this assignment on the users request. If one of the arguments of a command is prefixed by ">" the shell will change the standard assignment of the output file to the file* named after ">". Thus

```
> filex
```

means "place output on filex" instead of "on the standard output device (i.e. the terminal)". The symbol "<" has the corresponding meaning for the standard input.

If a command is followed by an "&" the shell will not wait for this command to terminate but will start a parallel task for the execution of this command. In order to identify this new task, the shell will return with a task identification number. It is thus possible to start background processing. The scheduling algorithm* in UNIX+ is designed to give good response to the interactive foreground process in case the background process is very processing intensive.UNIX+ also provides special facilities for the communication between parallel tasks.

A more detailed description of UNIX+ can be found in the UNIX+ books /Bourne, S.R./, /Banahan et al/ in the bibliography.

Appendix 2

SADT# -- Structured Analysis and Design Technique /Ross 1977/

SADT# is a comprehensive methodology for systems analysis and functional design.

It has been developed by the company SOFTECH in Waltham, Mass. USA in the seventies. The principle author is D. Ross.

SADT# provides techniques and methods for:

- thinking in a structured way about large problems
- representing the results of the analysis and design* phase
- communicating the results in a clear notation
- team work in the analysis and design* phase
- managing the analysis and design* phase

SADT# is based on seven Fundamental Concepts

(1) Understanding of Systems via Model Building

The in-depth understanding of a system is achieved by building "models" of system from well defined viewpoints. Such a model is an abstract representation of the system, eliminating all details which are unimportant for this specific viewpoint. Different Aspect of the system will be represented by different models, e.g. a Model may describe the Functional Characteristics of a System, another model may be concerned with the Maintenance Characteristics.

(2) Top Down Decomposition

Any SADT# Model is developed from outside in. The Top level is concerned with a complete, but very general description of the system. At each level down, the concepts of the previous level are refined and more details are brought in. SADT# limits the amount of additional information that may be brought in at any one level.

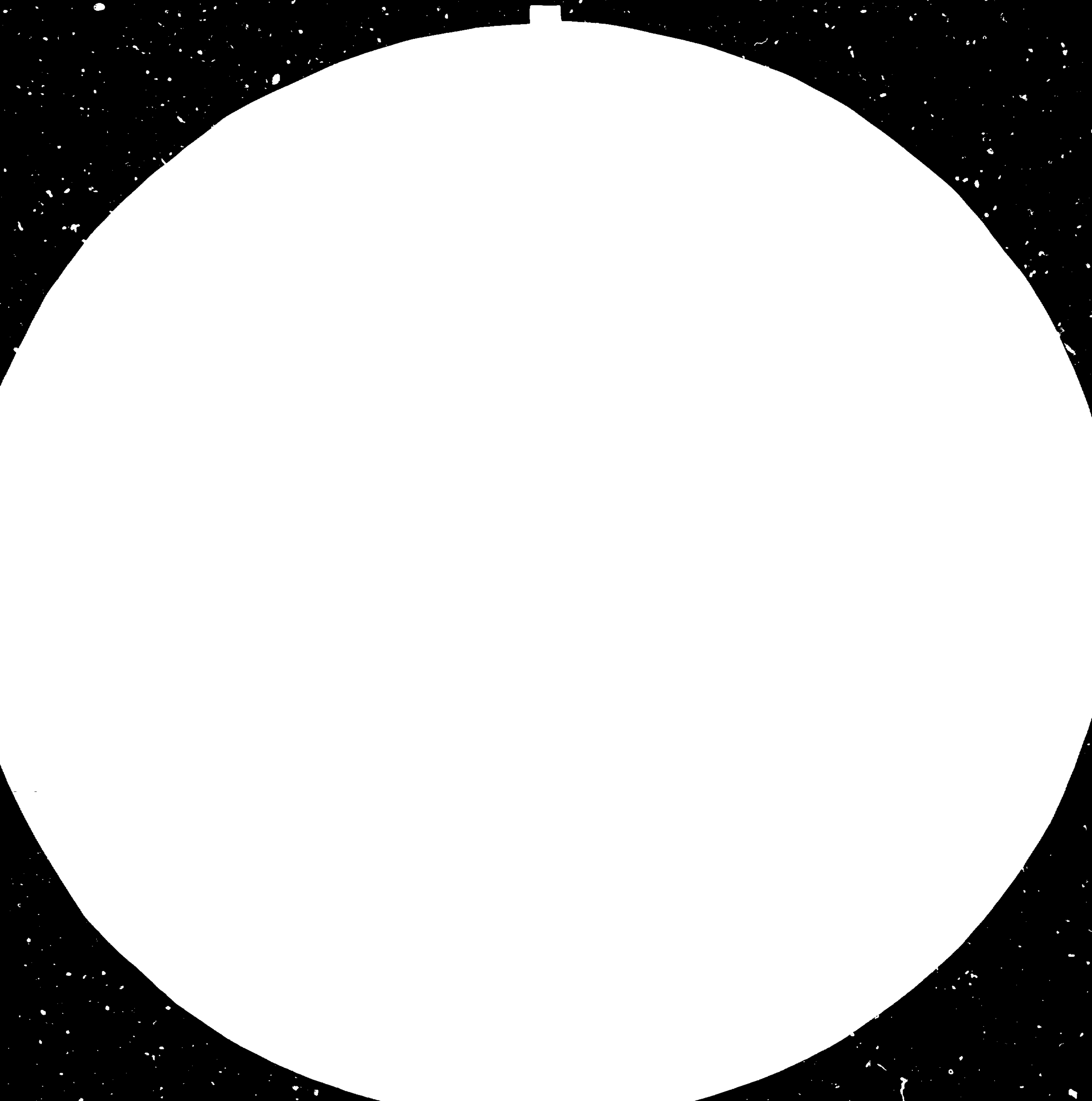
(3) Functional Modeling versus Implementation Modeling

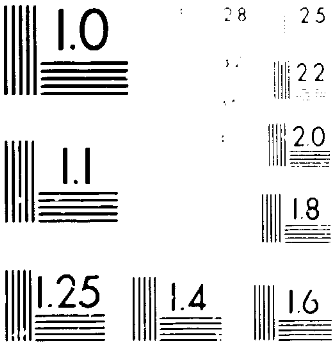
The starting point is always a Functional Model of the problem: "What is it?" as opposed to "How is it implemented?" . The development of a clear and precise functional specification before implementation is of critical importance to successful system production. SADT# provides a notation distinguish between a function and a mechanism used to implement this function. Sometimes a mechanism may be so complex that it in itself warrants to development of its own model.

(4) Dual Aspects of Systems

System may be described in many different ways. SADT# distinguishes between two methods of looking at a system -- its entities (data) or its activities (processes). The corresponding models are called a data* decomposition and an activity decomposition. The data decomposition details the "things" of the system, while the activity decomposition details the "processes".
In the final phase of modeling a correspondence verification of these two decompositions has to be performed.

84.04.12




1.0
2.8
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(5) Graphic Format for Model Representation

SADT# provides a graphical language* to represent the analysis and design. The main elements of this graphic language* are boxes and lines to connect the boxes. The number of elements on a diagram is strictly controlled in order to prevent overloaded diagrams. The interpretation of the boxes and lines between the boxes is different for the "datagram",i.e the data* oriented design* and the "actigram", i.e the activity oriented design. In datagrams, the boxes refer to data* elements and the lines to the activities producing and consuming these data* elements. In actigrams, the boxes refer to activities and the lines to the input and output data* of these activities

(6) Support of Disciplined Team Work

The Analysis of Complex systems requires the cooperation of many people. SADT# provides a set of rules for such a team work. Each member of a team has to conform with the role assigned to him. Among the different roles in the teamwork we distinguish between

- the author who actually writes the SADT# diagrams
- the reader who has to read and interpret the SADT# diagram
- the expert who has to provide the information about the system
- the secretary who has to record and file* all information
- the monitor who has to monitor the progress of the SADT# project and look after the conformance with the SADT# rules.

(7) All Decision and Comments in Written Form

In SADT‡ all decisions and alternate approaches have to be recorded in written form. Authors have to write the SADT‡ diagrams and experts have to comment on these diagrams in written form. The diagrams have to be filed with the secretary. This complete documentation of a project helps to clarify misunderstandings and reduces the number of iterations.

There are some software packages available which provide computer assistance for the SADT‡ user.

Appendix 3 :

Glossary of some software terms

Most of the following definitions of software terms are taken from IEEE St. 729-1983 "IEEE Standard Glossary of Software Engineering Terminology", published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA Explanatory remarks are added.

ADA++: A new programming language for real time applications
see also chapt. 2.3

algorithm: A finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, the set of rules which have to be followed in the solution of a mathematical equation.

application software: Software specifically produced for the user of a computer system; for example, a payroll program or a program for the control of a specific machine. Contrast with system software.

assembly language: A machine specific language whose instructions usually in one-to-one correspondence with the hardware instructions of the computer.

BASIC: A programming language which is simple to use, see also chapt. 2.3

C : A programming language for systems programming, mainly in UNIX+, see also chapt. 2.3

COBOL: A programming language used for commercial programming, see also chapt. 2.3

change control: The process by which a change to the software is proposed, evaluated, approved or rejected, scheduled, and tracked.

command language: A set of procedural operators with a related syntax, used to indicate the functions to be performed by an operating system. Synonymous with control language.

comment: Information embedded within a computer program command language or a set of data that is intended to provide clarifications to human readers and that does not effect machine interpretations.

compile: To translate a higher order language program into a form which can be executed by the machine. The corresponding translation program is called a compiler. Contrast with assembler, interpreter.

computer: A functional programmable unit that consists of one or more associated processing units and peripheral equipment, that is controlled by internally stored programs, and that can perform substantial computations, including numerous arithmetic operations or logic operations without human intervention.

concurrent processes: Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrence of an external event.

data: a representation of facts, concepts or instructions in a formalized manner suitable for communication, interpretation or processing by human or automatic means.

data communication protocol: A set of rules defining the data structures and the duration between events for the communication between computers

design: The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements. Also: the results of the design process.

efficiency: The extent to which software performs its intended functions with a minimum consumption of computing resources.

embedded computer system: A computer system that is integral to a larger system whose primary purpose is not computational; for example, a computer system in an aircraft control system.

execution: the process of carrying out an instruction of a computer program by a computer.

failure: the termination of the ability of a functional unit to perform its required function.

fault: An accidental condition that causes a functional unit to fail to perform its required function.

file: a set of related records treated as a unit.

FORTRAN: A programming language for scientific applications see also chapt. 2.3

functional decomposition: A method of designing a system by breaking it down into its components in such a way that the components correspond directly to system functions and subfunctions.

functional specification: A specification that defines the functions that a system or system component must perform.

hardware: Physical equipment used in data processing as opposed to computer programs, procedures, rules and associated documentation. Contrast with software.

interface: a shared boundary between two or more subsystems or a system and its environment. A specification that sets forth the interface requirements is called an interface specification.

language processor: A computer program that performs such functions as translating, interpreting, and other tasks required for processing a specified programming language; for example a FORTRAN processor, a COBOL processor etc..

LISP: A programming language for Artificial Intelligence applications, see also chapt. 2.3

machine language: a representation of instructions and data that is directly executable by a computer.

PASCAL: A programming language for teaching programming
see also chapter 2.3

procedure: a portion of a computer program which is named and which performs a specific task

project plan: A management document describing the approach that will be taken for a project. The plan typically describes the work to be done, the resources required, the methods to be used, the schedules to be met and the procedures to be followed.

Program: The instructions which tell the computer what has to be done.

Programming language: An artificial language which can be used to express the instructions to a computer

PROLOG: A programming language for artificial intelligence applications, see also chapter 2.3

Protocol: see "data communication Protocol"

requirement: A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document. The set of all requirements froms the basis for subsequent development of the system or system component.

specification: A consise statement of a set of requirements to be satisfied by a product, a material or process indicating, whenever appropriate, the procedure by means of which it may be determined whether the requirements given are satisfied.

security: The protection or computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure. Security also pertains to to personnel, data communications, and the physical protection of computer installations.

software: Computer programs, procedures, rules and associated documentation and data pertaining to the operation of a computer system. Contrast with hardware.

software documentation: Technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain the desired results from a software system.

software life cycle: The period of time that starts when a software product is conceived and ends when the product is no longer available for user.

source program: A computer program that must be compiled, assembled, or interpreted before being executed by a computer.

system software: Software designed for a specific computer system or family of computer systems to facilitate the operation and maintenance or the computer system and associated programs; for example, operating system, compilers, utilities. Contrast with application software.

testing: The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Bibliography

/Banahan 1982/ Banahan, M.F.. Rutter, A., UNIX+ — the Book Sigma Technical Press, Wilmslow, U.K., 1982

/Boehm 1981/ Boehm, B.W., Software Engineering Economics Prentice Hall, Inc., Englewood Cliffs, N.J., 1981

/Bourne 1982/ Bourne, S.R., The UNIX+ System, Addison Wesley Publishing Company, London, 1982

/Buckle 1982/ Buckle, J.K., Software Configuration Management MacMillan, London, 1982

/Collins 1982/ Collins, G., Blay, G., Structured Systems Development Techniques, Pitman, London, 1982

/Dunn 1982/ Dunn, R., Ullman, R., Quality Assurance for Computer Software, Mac Graw Hill, New York, 1982

/Kopetz 1979/ Kopetz, H, Software Reliability, MacMillan, London, 1979

/Musa 1983/ Musa, J.D., editor, Stimulating Software Egnineering Progress, A Report of the Software Engineering Planning Group, in Special Issue of software Engineering Technical Committee Newsletter, Vol.7, No. 4 p. 1-26, May 1983

/Myers 1979/ Myers, G.J., The art of software testing Wiley Interscience, New York, 1979

/Ross 1977/ Structured Analysis: A Language for Communicating Ideas, IEEE Transactions on Software Engineering, Vol. SE 3, No. 1, Jan. 1977, p.16-34

/Shooman 1983/ Shooman, M.L., Software Engineering, McGraw Hill Book Company, New York 1983

/Simon 1971/ Simon, H.A., The theory of problem solving, Proceeding of the IFIP Congress Ljubljana, 1971, p.I.249 -266

/Somerville 1982/ Sommerville, I., Software Engineering, Addison Wesley, Reading Mass, 1982

/Wassermann 83/ Wassermann,A, Porcella, M., Freeman, p., "ADA++ Methodology Questionnaire Summary", Software Engineering Notes, Vol 8, No. 1, Jan. 1983, p.51 - 99

/Wolverton 1972/ Wolverton, R.W., The cost of Developing Large Scale Software, IEEE Trans. on Computers, June 1974, pp.615 - 636.